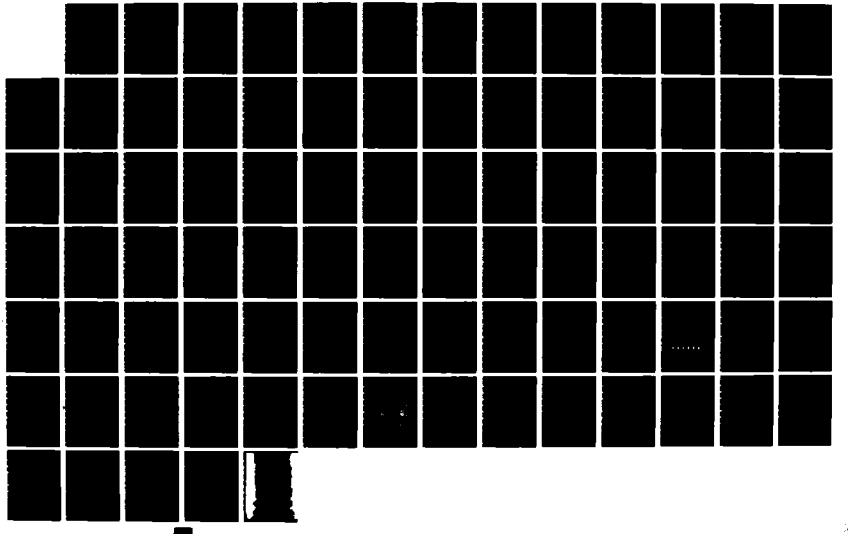
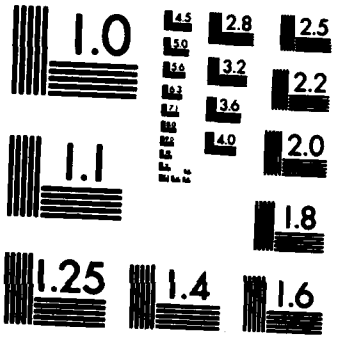
AD-A142 959 PRINCETON VLSI PROJECT(U) PRINCETON UNIV NJ DEPT OF 1/1
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON
1984

UNCLASSIFIED F/G 9/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PRINCETON VLSI PROJECT: Semi-Annual Report

PERIOD ENDING: March 26, 1984

Richard J. Lipton - Principal Investigator

EECS Department
PRINCETON UNIVERSITY

FACULTY:

Bruce W. Arden, Chairman
David P. Dobkin
Hector Garcia-Molina
Peter Honeyman
Andrea LaPaugh
Kenneth Steiglitz

PRINCETON VLSI PROJECT

B. Arden, D. Dobkin, H. Garcia-Molina,
P. Honeyman, A. LaPaugh, R. Lipton, K. Steiglitz

1. Introduction

There are three major components to the project. The first is in the area of procedural design of VLSI circuits. The second is in the area of our census language, and the third is in the area of the testing of VLSI circuits.

2. Procedural Approach to VLSI Design

2.1. ALI2 [LaPaugh, Mata]

ALI2 has been operational for a number of months now. A variety of chips have been designed and fabricated using ALI2; testing of them is now underway.

Already work is under way to improve ALI2 and make it more powerful. The major new idea here is based on a new algorithm for solving a more powerful class of constraints that we could handle before. This new algorithm allows the designer to completely mix both rigid objects and flexible ones: previously we could only allow flexible objects. We feel that this algorithm with its excellent running time may have applications beyond just ALI2.

2.2. Clay [Lipton, North]

Clay our other procedural language is also now operational. Chips designed with it our now just returning from fabrication and our being tested. Clay is now at a number of other institutions. For example, Freeman at Brown has already made extensive use of Clay and has already added a number of features to Clay such as a trace package. He is also planning to add a smart leaf cell generator to Clay as part of his PhD work at Brown.

North has also begun to think about building a better graphics interface between Clay and a bit map display. Ideally, we would like to allow the free mixture of pictures and programs: we believe that such a mixture would be a very powerful way to express complex layouts.

2.3. Applications of Clay

2.3.1. Graphics Engine [Dobkin, Field, Souvaine]

The current goal is to build high performance engines based on pseudo-triangles. Various parts of the design have been completed in Clay and are now in the process of being fabricated.

2.3.2. Recursive Layout [Steiglitz]

A number of recursive layouts have been completed and are now off to fabrication. Such recursive layouts are especially easy in a procedural language such as Clay. Recursion is a powerful way to design regular structures such as those found in digital signal processing.

2.3.3. PRISM [North]

PRISM is a special purpose processor that is highly optimized for non-numeric computations. It includes a large address and a large data stack of 4k words. We are currently designing it as a bit-sliced set of chips. We feel that this bit-sliced structure is exactly what is needed to later on put PRISM onto a wafer-scale structure.

In addition, PRISM includes a yield-enhancement mechanism. It includes a simple way via settable switches to de-select parts of its large stack. This will, of course, greatly improve the yield of the chip. We are about to fabricate the first PRISM chips and will then be able to determine the exact yield improvement obtained.

3. Census

There are two main projects under way here.

3.1. Top/Down [Lopresti, North]

This project is investigating the use of census approach to parallel computations. We have a four processor system now running: each processor is a Motorola 68000 with 256K memory. A large number of experiments are under way to test out the top/down approach to parallel computation. So far we have mainly run "simulated annealing" type computations for problems in PLA state assignment, routing, and placement. We plan shortly to expand the machine to include several more processors.

3.2. MMM [Garcia-Molina, Honeyman, Lipton]

This project is investigating the Massive Memory Machine. We have begun fairly detailed space simulations of a variety of computations. These simulations are collecting a variety of statistics directly from running programs. For example, the "time" command has been locally modified at Princeton to return the high

water mark or maximum space used by the executing process.

4. Testing [LaPaugh, Steiglitz, Vergis]

Work continues on a variety of approaches to VLSI testing. For instance, Vergis's PhD will contain a powerful set of methods to allow the efficient testing of certain systolic arrays. These results greatly generalize old results on the testing of simplier structures.

5. Papers

A Top-down Approach to Parallel Computation

Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

Stephen C. North

AT&T Bell Laboratories
Murray Hill, NJ 07974

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

ABSTRACT

We consider a top-down approach to parallel computation by parallelizing a sequential computation at the highest level possible. This approach offers the advantages of simplicity, generality, ease of programming, and tolerance to benign hardware failures.

March 21, 1984

Supported in part by DARPA N00014-82-K-0549

A Top-down Approach to Parallel Computation

Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

Stephen C. North

AT&T Bell Laboratories
Murray Hill, NJ 07974

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

1. Introduction

Once a computer program is working the next major concern is that of performance: can we make it go "faster"? There are many approaches to this important problem. First, we can often achieve tremendous speedup by selecting a new data structure or a better algorithm. For example, a program that repeatedly searches a set of objects may be vastly improved by using a hash table instead of linear search. Second, we can often achieve just as substantial speedups by careful tuning of the chosen data structures and algorithms. Usually this "hacking" is not done on the whole program, but is applied to the "inner-loop" of the program [1]. It is frequently observed that programs spend most of their execution time in a very small fraction of their code which is iterated many times. This method of speedup based on hacking the inner-loop of programs we call the ***bottom-up*** approach.

A third way to speed up a program is to map it onto a parallel machine. Today there is a great deal of research into the exploitation of parallel machines as a way to speed up computations. This is a desirable goal since advances in technology have made processors inexpensive. Our central thesis is that most of this research makes an incorrect assumption:

> They implicitly assume that the only way to achieve speedup on a parallel machine is to take the bottom-up approach, i.e. to parallelize the inner-loop of the given sequential computation.

Our fundamental point is that this is not the only way to achieve great speedups with parallel machines.

We propose that an alternative approach, which we call *top-down*, is often a more powerful way to achieve speedups. Note that often the inner loop consists of many independent evaluations of the same function. Since our goal is really to speed up the execution of the entire program, it is not necessary to speed up the inner loop itself. Instead, the top-down approach obtains the speedup simply by running the independent iterations in parallel on many processors. To parallelize the solution of a problem, we partition it into independent subproblems. Each subproblem can be solved by a set of independent executions of the inner loop on one of the parallel processors.

For instance, a Monte Carlo simulation computes the value of some function at many different random points. These simulations are are usually processor intensive, and in some cases researchers have resorted to designing special-purpose hardware to obtain acceptable computational speed [2]. A bottom-up approach to speeding up this simulation on a parallel computer would focus on making the evaluation of the function at each individual point faster, by using parallel processors in some clever way. However, none of the function evaluations depends on any previous one. The top-down approach, then, is to assign each parallel processor its share of the set of random points and let it compute the function on this set, independently of the other processors.

Our model for parallel computation is a large number of processors connected by a very general network, such as an Ethernet or high-speed parallel bus. This network can be connected to a host mainframe, or one of the processors can be designated as a coordinator. To parallelize a sequential program, we break up its set of input data into subproblems of equal size Then the program

is broadcast to all the slave processors, and each is also sent its subproblem. The slave processors then run the sequential program on their sets of data, and send results back to the host. For this approach to be effective, computation must dominate input/output. Fortunately, this is the nature of the compute-bound programs we wish to speed up.

More formally, express the function to be computed as:

$$g(f(x_1), f(x_2), \cdots, f(x_n))$$

The top-down approach is to parallelize the computation of the f by distributing the x_i among the processors. The coordinating processor computes their composition, g. g is usually an easily computed function, such as taking the min, max, or average of the f.

The top-down approach will not allow us to parallelize all sequential computations, such as those where each execution of the inner loop depends on the result of a previous execution, or where a suitable decomposition does not exist. Nevertheless, the domain of practical problems which have top-down decompositions is very large. (Some examples will be considered in section 3.) We feel that the simplicity of the top-down approach is of great value, and that it has been overlooked in the past, in favor of trying to solve the interesting problems caused in part by the complexity of bottom-up parallel computing.

2. Advantages of Top-Down Parallel Computation

1. ***Ease of programming***. There is already a wealth of experience in sequential programming. Consequently, sequential algorithms are easier to design and to implement than those with bottom-up parallelism. Using a bottom-up approach, one must carefully learn the properties of the parallel computer to exploit it successfully. Compilers to generate good code exploiting the parallel computer and other important tools may not even be available. Also, in some cases, an entirely new parallel algorithm must be invented to use

the parallel computer, which may require radically rethinking the problem By contrast, with the top-down approach, existing sequential algorithms can be run almost without change. New algorithms do not have to be invented Also, the program can be debugged on a single processor before running it in parallel, which simplifies the debugging task.

2. ***Flexibility and generality***. The top-down approach is ideal for many existing architectures and networks. A machine such as we have described can be built with off-the-shelf hardware or software, or using a local area network of personal computers. Since the hardware is not designed around the algorithms, improvements in the sequential algorithm run by each machine will not make the hardware obsolete. On the other hand, a bottom-up approach to parallel computing often requires special-purpose processors or an interconnection network that is highly adapted to running a particular class of algorithms, or even solving a given problem using a given particular algorithm. If a different problem is to be solved, or a better algorithm is discovered, the special-purpose machine may not be able accommodate it. Also, the top-down parallel machine is easily expanded by simply adding more processor boards to the network. Thus the user has great flexibility in choosing the cost and performance of the parallel computer.

3. ***Low communication costs***. Most computation is done locally on each processor, which does not need to communicate with its neighbors. The only communication required is for the controlling processor to send the program and data, and for the slave processors to send results. Therefore, interprocessor communication does not become a bottleneck limiting the speed of the computation. No special network topology is needed.

4. ***Linear speedup***. If the parallel computer has p processors, we can expect an almost p-fold speedup over the execution of the same algorithm on a single processor, assuming that the costs of computing the f are relatively uniform and the costs of f dominate the cost of g. The improvement in

performance with the top-down programming technique is both good and predictable.

5. ***Fault tolerance***. A top-down design is potentially more fault tolerant than a bottom up design which needs all its processors to be functional. We assume that hardware failures in the slave processors are benign, that is, they are detected when they occur by causing an interrupt in the failing processor or by preventing its operation altogether ("fail-stop"). For some tasks, not all but only a large fraction of the $f(x_i)$ need to be computed. This is the case with Monte Carlo simulations. If a slave processor fails, its output is discarded. Another strategy is to reserve a few slave processors as spares. The coordinator can periodically poll the slaves to test if they are still functioning. When a failure is detected, the failing processor's work can be reassigned to a spare. The time lost because of the failure will depend on the size of the independent subproblem that must be reassigned. This scheme is not completely immune to hardware failures, but if properly designed a top-down parallel computation can be more fault-tolerant than a bottom-up scheme which requires all the processors to be functioning at once.

3. Examples

We will briefly describe a few possible applications for the top-down approach to parallel computing.

1. Simulations and Exhaustive Searches. We have already described how integration by Monte Carlo simulations maps very cleanly onto our model of parallel computation. This approach will also work when we need many independent runs of the same simulation to solve a nonlinear optimization problem. For such problems, we have a set of parameters for a complex model and an objective function whose value we wish to optimize. To find good though not necessarily optimal solutions to such problems, we can try many randomly chosen parameter values and compute the resulting value of the objective function. We

can liken this process to turning the knobs on a black box with a meter on it, trying to obtain the highest reading. By trying many random values, we hope to find a good solution. Note again that the random trials are independent. So each $f(x_i)$ is a parameterized simulation, and g is an easily computed function, such as taking the min or max of the f. Further, such a simulation can be fault-tolerant, since the failure of any single processor results only in the loss of a small fraction of the random trials. Similar techniques, such as the use of random trials with hillclimbing to find local optima, are valuable for finding approximate solutions to combinatorial problems [3].

Counting problems, such as enumerating graphs with some given property by generating graphs in a regular pattern and then testing them [4], as well as exhaustive searches of other regularly generated combinatorial objects, including game trees, can also be solved top-down.

As another example, consider the problem of boolean circuit simulation for VLSI fault testing. Given a circuit c, we wish to find a set of test vectors that will uncover all possible faults under some model. Let $E=\{e_i\}$ be the set of possible faults. Let $V=\{v_i\}$ be the set of input vectors to c. The set of test data will be some $W=\{w \mid w$ is an input to c and w_i detects a fault in $E\}$. Further, we want W to be complete, so that every fault in E is covered, and to be small. To find a set W, our algorithm will generate candidate input vectors w_i and simulate both c and some $e_j \in E$. If they can be distinguished, then w_i is added to W and e_j is removed from E.

There are two ways to parallelize this algorithm. First, we can distribute the set of faults E across processors and let each exhaustively try members of V until all its members of E are eliminated. The slave processors then report the set of test vectors they found back to the host. To improve performance we might have the host keep track of which elements of E are currently "alive," and once a minute redistribute E to balance the work load. Second, we can partition V and broadcast E to all the processors. Each slave tries its subset of V,

and remembers those which eliminate members of E. Again the host can keep track of the live elements of E and occasionally broadcast this information, so that if one slave finds an inputs that detects a fault, other slaves stop working on it. In fact, the first strategy is better at the beginning of the simulation when the set of faults is large and most of them are easily killed off. The second strategy is better later in the run, when there are only a few faults left but they are difficult to detect. Therefore both strategies could be used at different times in the program execution.

2. Design rule checking of VLSI layouts is another candidate for top-down decomposition. The input and output for this problem are comparatively small—the input is usually a description of the polygons composing the layout; the output indicates which (if any) are incorrectly placed. However, a great deal of computation is required. Design rule checking can be parallelized by decomposing the layout into sections, which are checked independently. A slight overlap is needed between adjacent layout sections to detect errors at the boundaries. Tom Lane at Carnegie-Mellon has implemented a portion of this algorithm, counting intersecting rectangles, on the Cm* parallel computer, and obtained a 15-fold speedup with 20 processors [5].

3. Animation and image processing. Producing high-quality animation sequences using ray-tracing requires much computation per frame. However, it is often the case that once the basic animation sequence is defined, each frame can be processed independently.

4. "Pipelined" realtime computations. Consider the case where we wish to process a stream of data in real time, such as digitized images coming in from an orbiting satellite for image analysis or pattern recognition, or to compute Reed-Solomon codes for deep space communication. In the latter case, systolic VLSI arrays have been proposed to provide fast enough response to the incoming stream of data, since conventional microprocessors are too slow. However, if we only require that the parallel computer keep up with the stream of data, and not

that it provide actual realtime response, then we can send each processor a portion of the input as it arrives, and accept the processor's output later when it is ready, in a round-robin schedule. For instance, if a new image arrives from a satellite every second, and it takes five minutes for a microprocessor to process an image, then a computer with 300 microprocessors is needed. There will be a five minute delay for processing of an image, but images will be output every second.

5. Rule-Based Systems. These artificial intelligence systems consist of a global state, and a set of rules with predicates. The system searches through the predicates to find one which is satisfied by the current state, and then applies the corresponding rule to obtain a new state [6,7]. The expected performance of such a system is from only a few rule firings per minute to upwards of several thousand per second. This performance is still considered inadequate for many expert system designs. To parallelize searching rules in a production system, we could assign each processor a fraction of the rule base. The coordinating processor broadcasts updates to the current state. Each slave processor then examines the predicates of its set of rules to determine which can fire, using the sequential algorithm, and sends the rule back to the coordinator. The coordinating processor can arbitrate in the event that multiple rules fire. We thus obtain a linear speedup in the search.

4. Conclusion

The top-down approach we have outlined has important advantages for speeding up sequential programs. It can be applied to a wide range of multiprocessor hardware. Although future research in parallel computing can be expected to uncover automatic techniques for finding top-down decompositions, at present many practical problems can easily be parallelized by hand.

5. References

[1] Bentley, J. ***Writing Efficient Programs***. Prentice-Hall, 1982.

[2] Pearson, Robert B., John L. Richardson, and Doug Toussaint. A Fast Processor for Monte Carlo Simulation. ***Journal of Computational Physics***, 51, pp. 241-249 (1983).

[3] Papadimitriou, C. H., and K. Steiglitz. ***Combinatorial Optimization: Algorithms and Complexity***, pp. 454-486. Prentice-Hall, 1982.

[4] Robinson, R.W., and N.C. Wormald. Numbers of Cubic Graphs. ***Journal***

[5] Lane, T. Carnegie-Mellon University. Personal communication, 1/31/84.

[6] Forgy, Charles L. ***OPS5 User's Manual***. Dept. of Computer Science, Carnegie-Mellon University. of Graph Theory Vol. 7, No. 4, pp. 463-467.

[7] Clocksin, W.F. and C.S. Mellish. ***Programming in Prolog***. Springer-Verlag, 1981.

THE CASE FOR MASSIVE MEMORY

Hector Garcia-Molina
Richard Cullingford
Peter Honeyman
Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N. J. 08544

ABSTRACT

We argue that for certain important classes of non-numeric computations, memory is more of a critical resource than computing cycles. Therefore, we postulate that there is a need for a computer with truly massive amounts of primary storage, on the order of billions of bytes. We believe that such a machine, even with a relatively slow processor, can outperform all other supercomputers on memory bound computations. This machine would be simple to program. In addition, it could lead to new and highly efficient programs that trade the available space for running time.

This work was partially supported by DARPA grant #N00014-82-K-0549.

March 21, 1984

THE CASE FOR MASSIVE MEMORY

Hector Garcia-Molina
Richard Cullingford
Peter Honeyman
Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N. J. 08544

1. INTRODUCTION.

In recent years, the quest for so-called *supercomputers* has intensified dramatically. Much of the current wave of interest was sparked by the Japanese Government's *fifth generation* computer project, which promises to deliver extremely powerful computers in the near future. But of course, the real driving force behind the Japanese and other supercomputer research efforts is the large collection of very important problems that cannot be solved efficiently on today's computers.

Most of the supercomputers that are being developed or investigated today rely heavily on parallelism for their processing power. They either have large numbers of processors, capabilities for parallel vector operations, are highly pipelined, or have combinations of all these features.

In this paper we argue the case for an entirely different type of supercomputer, one that bases its power not on massive parallelism, but on massive amounts of primary memory. We do not have a specific target size for such a *massive memory machine* (MMM), but for argument's sake let us say we want on the order of *tens of billions* of bytes of main physical memory. This size is certainly larger than is offered by any manufacturer today, or is likely offer in the near future. Our thesis is that a MMM is justified, even today, by the importance of certain applications in which *memory bound computations* occur naturally. For these computations, a parallel supercomputer will be severely limited by the

rate at which it can transfer data in and out of its memory. Therefore, a classic von Neumann machine, even with a relatively slow processor, but with massive amounts of physical memory, would vastly outperform any parallel supercomputer on these problems and would be, in addition, far easier to program.

In this paper we are not proposing a novel computer architecture, nor are we claiming to have the ultimate supercomputer. We are simply putting forth an idea that that appears to have been overlooked: for certain applications (and several of these will be described shortly), memory is a more precious resource than computing cycles, and hence, there is a need for computers that have an abundance of the former.

In addition to making the case for a MMM, our paper has two related goals. The first is to convince computer manufacturers that they should increase the physical memory limits of their processors. It is somewhat distressing to see that it is currently very difficult to purchase more than 32 megabytes for a processor, at a cost of about 70,000 dollars, but at the same time, one can spend hundreds of thousands of dollars on a processor. A second goal is to spark interest in large memory computers, for there are a number of intresting research questions that must be answered before the truly massive memory machines can be implemented.

The MMM we propose has two characteristics that distinguish it from conventional and super computers and that give it its unique power. The first is a processor that, given the size of its memory, is relatively slow. In Sections 2 and 3 we argue that such a balance between processor speed and memory size is advantageous for many important memory bound computations. The second characteristic is a massive memory, and in Section 4 we argue that this will fundamentally change the way certain large problems are solved on a computer, and will lead to huge performance improvements. Finally, in Section 5 we briefly discuss some of the open research issues.

2. HIGH MEMORY SIZE TO PROCESSOR SPEED RATIO.

If we look at the ratio

$$\alpha = \frac{\text{memory size}}{\text{processor speed}}$$

of past and present commercial computers, we find that most are within an order of magnitude of one megabyte per MIPS. (A MIPS is a processor execution rate of a million instructions per second.) The value one megabyte per MIPS is sometimes called "Amdahl's constant." It is not entirely clear why commercial machines have stayed close to this value, but market forces appear to have played an important role.

The supercomputers currently being developed all have α ratios well below this value, and are targeted for computation intensive problems. For instance, some proposed supercomputers call for as many as one million processors, capable of executing billions of operations per second and yet have as "little" as sixty four megabytes of physical memory [Comp80, Comp81, Comp82, Evan82].

On the other hand, the machine we propose here is at the other end of the spectrum: it has an α ratio orders of magnitude larger than one. Why are we interested in such a machine?

First of all, it is not because we dislike fast processors. If we could have large memories *and* at the same time fast processors, we would of course take both. However, given our limited resources, we are investing a disproportionate amount in memory because this, not processing speed, is the major bottleneck for many non-numeric computations.

To illustrate this point, consider a program which accesses a four gigabyte (4×10^9 bytes) data structure with an essentially random pattern. Let us compare a supercomputer with one hundred megabytes of memory and a MMM with four gigabytes of memory. Further, let us assume that the supercomputer is "infinitely fast" while the MMM runs at only one MIPS. Of course the supercomputer will vastly outperform the MMM on compute-bound tasks. However, for the memory-bound

program we are discussing, assume that the supercomputer creates a page fault every f instructions, and that its disks are capable of servicing 100 requests a second. Then on this task the MMM still computes at its one MIPS rate while the supercomputer is reduced to computing at about $100f$ instructions a second. Clearly if f is small enough the MMM will be faster than the supercomputer: if f is about 100 then the speed advantage is 100:1! While not all tasks will cause the supercomputer to "thrash" in this way, we believe that there are a large collection of important tasks that will cause such behavior. And on these tasks, adding memory to a computer makes much more sense than adding processor power.

There is also economic evidence suggesting that high memory size to processor speed ratios will be advisable. Over the past few years, the price of logic circuits has decreased about 20% per year, but during that same span, memory prices have decreased at twice that rate: almost 40% per year [With83]. The main reason is that memories are highly regular integrated circuits, and thus, profit immediately from higher fabrication densities. This indicates that increasing the power of a computer through additional memory will be more cost effective than through faster processors.

3. APPLICATIONS.

There are many tasks that reference a large address space in a relatively random fashion, and for which memory is the critical resource. Here we review three areas in which such tasks abound, but this list is by no means exhaustive.

(a) **Databases**. It is well known that in many database applications, user requests are computationally very simple, yet require data from unpredictable locations in the database. Thus, a major portion of the response time to each user request comes from I/O waiting. Clearly, if the entire database or a substantial fraction could reside in main memory, then the I/O component would be reduced substantially,

possibly even eliminated.

The improved response time will be most valuable in real-time applications, but even in cases where users are willing to wait seconds for their answers, massive memory may have important advantages. Specifically, it may now be possible to pose interesting new queries that previously required unreasonable times to answer (e.g., a statistical query that requires one or two passes over the entire database). Thus, users can get more useful information out of the system.

(Reliability may be a problem in a massive memory database. We return to this and other database issues in Section 4.)

(b) **VLSI Design**. The size of VLSI circuits being designed is growing at a fast rate. Today there are circuits with a half million transistors, and predictions of integrated circuits with as many as one hundred million transistors by the mid 90's. VLSI design tools will perforce deal with massive amounts of data, notwithstanding much cleverness in the use of hierarchical design and the encoding of information.

Many of the VLSI design algorithms have good asymptotic running times, but have very poor locality of reference. Thus, they are naturally candidates for a high α machine. For example, a layout system we have designed [Lipt82] uses topological sorting for placing objects. The algorithm for sorting requires linear time, but unfortunately also requires linear space and has almost no locality. Thus, beyond a certain layout size, its actual running time is determined by the memory available: at a given point, increasing the layout size by 30% sends our computer into uncontrolled thrashing and increases the running time ten fold!

(c) **Artificial Intelligence**. The concept of vast data structures built mainly by the use of pointers, and hence lacking much locality of reference when accessed, immediately brings the words "*LISP*" and artificial intelligence (AI) to mind. Garbage collection [Cohe81] and paging contribute substantial fractions to the total running times of

many AI programs. It seems fair to say that a good fraction of AI research involves memory-bound computations.

Certain AI programs, such as DENDRAL [Buch78] or MACSYMA [Mart71], have succinct inputs and generally produce succinct outputs, and yet may build enormous intermediate data structures. These programs are even better suited to our machine. They would not even need to incur the overhead of loading the massive memory as a database or VLSI program would.

Although there have been numerous studies on the reference patterns of programs [e.g., Frey75, Siss68, Smit82, Spri72], very few have actually looked at the *data* references of the memory intensive programs we are interested in. The one exception we found supports our thesis that memory is a critical resource. This paper, by D. W. Clark [Clar79], analyzes in detail the data references of three "real" LISP programs: a chemical structure generator, a parser for a speech understanding system, and a program that builds and executes partially ordered plans of action. In summary, Clark discovered that the programs do have substantial locality of reference. For example, between 85 and 95 percent of the references fall within the most recently accessed page (512 bytes). So even if we only had a single data page at a time in memory, the miss ratio, i.e., the probability of having to fetch the referenced data from secondary storage, would be 0.15 to 0.05.

However, if we assume, as Clark does, that a reference to secondary memory takes about 5000 longer than a reference to primary memory, we clearly see that this "low" miss ratio gives very poor performance. The solution is, of course, to keep more pages in main memory. Unfortunately, the miss ratio decreases slowly as more pages are kept in memory (with a LRU replacement strategy). To obtain a miss ratio of 0.001, 40 percent of the total data space for one program, and 80 percent for another program, must be resident in memory. (The miss ratios of the third program are not reported in the paper.) And because these

programs are memory intensive, even this miss ratio of 0.001 slows down the programs by roughly a factor of 6, as compared to a program that had all of its data in memory. This clearly illustrates that for these programs it is more effective to purchase memory to hold a substantial fraction of the data space, than it is to purchase a faster processor.

To be fair, we should mention that there is a second way to improve the performance of memory intensive programs, in addition to simply obtaining more memory: this involves restructuring the data to improve locality. However, we do not believe this to be a valuable alternative in general. In many cases it is simply impossible to get locality. To illustrate, consider a database that contains data on departments and the employees who work in them. If we place the employee records close to (i.e., on the same disk page) as the record for their department, we get good locality when we access a department and its employees. However, if we need a list of all departments, then we have to visit many pages. On the other hand, if we place all department records close together, we can find all departments quickly, but now finding a department and its employees takes longer. In other words, unless we expect a single type of query, it is not possible to improve locality significantly.

Even if it is possible to improve locality, we feel that forcing programmers to analyze the reference patterns of their programs and to structure their data accordingly it is a step in the wrong direction. A good analogy can be drawn with virtual memory: it is clear that virtual memory is not necessary if we ask programmers to overlay their programs and data (i.e., have the programs explicitly state what resides in memory when). Furthermore, using overlays can be more efficient than using virtual memory (at least from the point of view of the computer). Yet, since overlays are so painful and difficult to use, we do not see many people who advocate their return.

Finally, it can be argued that what we really need are tools for *automatically* restructuring data to improve locality. This is certainly a good direction to pursue, but we do not expect such tools, if they ever

become available, to be useful for solving *general* problems. Incidentally, the paper by Clark describes one effort to automatically improve locality. The idea is to periodically compact the used data space, moving all the free space to a single area. Intuitively, it seems that this may help because it increases the density of data and increases the probability that a pointer leads to a nearby page. Unfortunately, as Clark reports, the effort to restructure the data is substantial (it involves traversing the entire data structure, writing it out to disk, and reading it all back in), and the improvements in locality are significant, but not great. (For example, the program that required 80 percent of its data to be resident to achieve a 0.001 miss ratio, now only needs 50 percent of its data resident; the program that required 40 percent, now only requires 35 percent.)

4. MASSIVE MEMORY.

In addition to having a very high memory size to processor speed ratio (α), the MMM we advocate (as its name indicates) has a massive memory. Why are we interested in a machine with billions of bytes of main physical storage?

The most immediate reason is that a massive memory will soon be economically feasible. Even at today's prices, the cost of the integrated circuits necessary to build a one gigabyte memory is below one million dollars. A complete computer may cost up to (roughly) 1.5 times this amount, but this is still not out of proportion with the investment necessary to equip state of the art installations for research or production work in some of the areas identified earlier. Furthermore, if the price trends hold [With83], by the end of the decade the same million dollars will purchase about 20 gigabytes.

Obviously, large memories are desirable because they increase the power of our machine: the larger the memory is, the larger the VLSI circuit that can be simulated on it, or the faster the LISP program will run

However, there is an even more compelling and exciting reason to go to massive memories: there are certain very important problems that have large, but bounded storage requirements. As soon as the main memory of a computer exceeds these storage requirements, the solution strategy for the problem changes entirely. With all the necessary data in memory, very efficient and simple techniques can be utilized, and this will lead to dramatic performance improvements, much larger than those which high speed or parallel processors by themselves could produce.

To illustrate this, let us first look at an application that does *not* require massive memory but that has already benefited from expanding memory sizes: text editing. In the vast majority of editing sessions, a user works on a relatively small document, on the order of tens of pages (e.g., a chapter or a paper). However, a few years ago main memories were so small that these documents could not fit in memory at once. Thus, text editors were designed to operate on very small portions of the document, i.e., a line of text. String searches over the entire document were avoided.

Now, of course, memories are much larger and can easily hold the text pages that a user is currently accessing. Modern editors utilize this memory to provide more efficient and easier to use systems. Screen editors let a user rapidly move from one page to the next, and may allow multiple windows for displaying different files. Bit-mapped displays, something that would have been unthinkable fifteen years ago, store an image of the screen in memory and make it possible to interactively change fonts and draw diagrams.

This example shows that having the data required by a program (e.g., the active text pages or the bit-map) in main memory can fundamentally change the way some problems are approached. In text processing this change has already occurred, but we now discuss some problems where massive memory is needed to bring about this transformation.

Transaction Processing. In a transaction processing system for airline reservations or banking operations, a high number of very simple *transactions* (e.g., to reserve a seat on a flight) must be executed. A typical transaction in these systems involves very few operations, and not counting system overhead, requires a few thousand instructions. Transactions are usually pre-compiled, and since there are a small number of transaction types, the code for them is kept in main memory. Yet, even with a 10 MIPS processor, the overall transaction rate will probably be low, on the order of tens of transactions per second.

The low throughput is due mainly to the delays encountered by transactions in reading and writing data from disk. However, there are other sources of overhead. Since transactions must wait for data, they are usually interleaved with other transactions. The concurrency control mechanism ensures that only interleavings that preserve data consistency are run. Not only does concurrency control add overhead, but the code for it is complex and elaborate. Similarly, data in main memory must be held in buffers, and managing and copying these buffers contributes to the overhead.

It is important to note that many of the databases used by transaction processing systems are within the 1 to 20 gigabyte range [Gray79] and are not growing nearly as fast as memory densities. For example, the number of accounts in a bank is usually limited by the number of residents in a state or country, and this number is not doubling every 2 or 3 years. Thus, it is reasonable to expect that many of these databases will fit entirely within the physical memory of a MMM.

Having the database in main memory radically alters the structure of a transaction processing system [Gray83]. With a main memory database, it is best to execute the transactions serially: all the data needed by each transaction are already in memory, and since the trarsactions are short, there is no reason to interleave. The database can be accessed directly, without need for buffers. Thus, in a system where transactions are pre-compiled and pre-loaded, most of the work

performed will be useful. If transactions take between 1,000 and 10,000 instructions, and the processor runs at 10 MIPS, we can expect execution rates between 1,000 and 10,000 transactions per second, several orders of magnitude higher than what is currently available on any system.

Even larger performance improvements can be obtained if we replace the database search structures, which are currently optimized for disk, by clever, new structures that exploit the available memory. For example, B-trees and indexed sequential files can be replaced with simpler, more efficient structures like hashing and binary trees. Updates to these structures will also be more efficient.

(Crash recovery is not simplified by a massive memory. [Garc83] addresses this problem in some detail, and shows how a massive memory database system can be made reliable without sacrificing the performance gains brought about by massive memory.)

AI Knowledge Bases. Many AI knowledge bases also have bounded size, and as memories grow, the databases will eventually fit in main memory. For example, a one gigabyte machine could hold one million "rules" of 1000 bytes each; this seems ample for an expert system in a specific domain (e.g., infectious diseases). Of course, if the system covers multiple domains, this may no longer be true. But for a single domain, a MMM can bring about dramatic improvements.

As with transaction processing, the improvements come not only because all the data is rapidly available, but also because new search techniques can be utilized. For instance, Cullingford and Joseph [Cull83] have developed a novel tree-like discrimination scheme for speeding up knowledge base searches that works especially well if all the data is in memory. The scheme automatically configures a discrimination tree that is heuristically arranged for maximum balance and bushiness. The size of the tree is potentially quite large (if the knowledge base to be searched is large), so that the scheme works best if all the nodes and predicates are in main memory.

A portion of the massive memory could also retain the most commonly made inferences. The system would consult this data (via fast hash table lookup) before each required inference. For example, an academic advisor program could directly tell engineering freshmen to enroll in section *b* of Math 101, without deducing every time that Math 101 is a requirement, but Math 101, section *a*, conflicts with Literature 100, another requirement, and section *c* this semester is for Math majors only. We can view this cache of commonly used facts as a very simple but effective learning mechanism that could bring substantial performance improvements.

Programming Environment. A massive memory could also lead to an improved programming environment, of the type currently implemented, with severe limitations, on some LISP, APL, and other systems [Fras83]. The main idea is that the user is not aware of the storage location of his objects, and simply sees a uniform "workspace" with a very large address space. All objects (e.g., strings, arrays, etc.) are located in this workspace, and can be manipulated with a single programming language. Thus, there is no need for a command language (e.g., UNIX Shell or CMS EXEC), and there is no need to save objects in files. (Of course, objects may still be arranged in hierarchies as in a file system, but the concept of "file" is no longer needed.)

The problem with current implementations of these ideas is that the active workspaces of a small community of users are larger than the few megabytes available on today's machines. Thus, thrashing occurs as the system struggles to get from disk the objects accessed by users.

The active workspaces of a small timesharing community are not growing very fast. For instance, the sizes and numbers of subroutines being debugged or of papers being edited are not changing much. Thus, as in our previous examples, it is reasonable to expect that a MMM will easily hold the active workspaces, will eliminate most disk accesses, and will make such systems much more usable.

Code Optimization. Our final example illustrates how massive memory can be utilized to hold pre-computed values that are likely to be referenced in the future, in order to avoid recomputing them. The code optimizer of a compiler takes sections of a program and produces optimum, or at least good, code for them. The most common strategy is to analyze each construct as the program is compiled. However, it is usually too expensive to compute the true optimal code, so a good approximation is produced for each construct.

With a MMM, a second strategy becomes feasible. It is now possible to precompute a large table that directly gives the best known code for the most common constructs. Since the table is only computed once, it does pay off to find the best possible code translations. Peter Wienberger at Bell Laboratories [Wien83] has experimented with this idea, and predicts that a few tens of megabytes would be necessary to hold the most common constructs in the C language, and their corresponding optimal code. As with our previous examples, the size of the table is not likely to change.

Thus, with a MMM, the optimization of a construct will in most cases simply involve a table lookup. Furthermore, the code produced would be superior to that which a conventional optimizer could produce in reasonable time. Of course, this strategy works best if the table can be kept in main memory. Since the table only takes a few dozen megabytes, this seems acceptable for a machine with several gigabytes of main memory.

Our examples have shown that there are a number of important problems that have large but slow-growing (or fixed) storage requirements. As larger and larger memories become feasible, these problems will succumb to massive memory solutions; solutions that are much simpler and orders of magnitude faster than what is currently available.

In closing this section, we stress three points:

Memory vs Parallel Processing. First, a massive memory machine and a parallel processing supercomputer address different, but equally

important problems. A MMM is no match for a supercomputer on computation intensive problems, and similarly, a supercomputer (without massive memory) cannot beat a MMM on memory intensive tasks. The following very simple example clearly illustrates this last idea. Suppose we have 10^7 records, with 100 bytes each, that fit within a MMM. To find a record given its key, we can construct a hash table. This memory-intensive strategy, on the average, would give us a record almost instantaneously, say, in the time to execute 5 to 10 instructions. It would be difficult for a supercomputer to beat this time. If the supercomputer does not have 10^9 bytes of main memory, the search will involve secondary storage and will clearly be slower. With enough memory and, say, 10^7 processors (with one record per processor), the supercomputer may beat the hashing time, but considering the synchronization overhead, the advantage may be slim. In any case, adding 9,999,999 processors is not cost effective!

Conventional Architecture. The second point is that a MMM, unlike a parallel supercomputer, has a conventional von Neumann architecture, at least from the user's point of view. (As discussed in the next section, a MMM may have an unconventional implementation that is transparent to the user.) This fact has important ramifications. It will be easy, if not trivial, for programmers to learn how to use the MMM. Existing programs could be run immediately on the MMM, and could profit from its large memory. (As we discussed earlier, new programming approaches may be even more profitable than the conventional ones, but the standard programs will still run, probably faster than on a conventional machine.) Just as important, existing support software like compliers and operating systems, could be used on the MMM. This again simplifies the user's learning task, and at the same time, reduces enormously the systems effort required to make the MMM operational.

Programming Style. Finally, the existence of massive memory might also drastically change the *style* in which large programs are written. Currently, many large systems tend to have a relatively large proportion of the address space used in program text (i.e., executable code), and

only a relatively small amount in data. For example, typical implementations of the UNIX operating system have about 70% in text and 30% in data. Large programs written this way tend to be hard to debug, maintain and understand.

An alternative style is a more ***data-driven*** or ***interpretive*** one. Here one writes a very simple control or monitor program which uniformly retrieves data objects which are themselves programs (written in a special language) or have program fragments attached to them. The controller then has the flavor of a ***dispatcher***, driven by the data it receives. Current AI rule-based [Shor76] and production-system [Wate78] techniques are also examples of a data-driven programming style. These approaches have not been popular heretofore because they tend to be slow with respect to a monolithic approach. But a large part of the cost lies in simply accessing, more or less randomly, the required data objects. This cost is high in a standard virtual memory architecture, but would be low in MMM.

The advantages of a data-driven style are simplicity of control, clarity and ease of incremental extension. One simply codes a new object in the simple, specialized language (a production-rule format, for example), and makes it available to the interpreter. Barring unexpected interactions with the other objects (which can be limited by the design methodology), the system simply runs with the new item. From the point of view of understanding a system designed this way, it is worth noting that a data-driven system can be made to ***explain*** its functioning. The data objects can be thought of as items of knowledge, expressible using any of a number of natural language generation techniques [e.g., Cull82].

5. OPEN QUESTIONS.

We hope the reader is convinced that memory is a critical resource in certain applications and that having large quantities of it may be useful. But is implementing a MMM simply a matter of purchasing the required memory and attaching it to a processor in a conventional way?

Are there any problems that must be solved before a MMM can be built? We believe that there are some challenging, but surmountable, problems. In this section we briefly outline some of them.

Bus Delays. As the number of devices on a bus increases, the memory access times also increase because of the physical distances and capacitance effects. Thus, as we go to larger and larger memories, we may lose part of the advantage of having a massive memory. There are a number of potential solutions to this problem, ranging from special purpose hardware (e.g., optical fibers), to sophisticated interconnections (e.g., a hierarchy of buses), to a novel computer architecture called ESP that is described in detail in [Garc84].

Reliability. There are two types of reliability problems. The first arises in any large computing system: as the number of components increases, so does the probability of failure. In the case of a MMM, we are fortunate in that the large number of components are memory cells. Memories are very regular structures, and it is relatively simple to add redundancy to them (i.e., error detecting or correcting codes [Siew82]) in order to reduce the probability of an error.

A second problem is that memory, unlike disk storage, is volatile. Thus, in applications where the data itself are important (e.g., databases) we must take additional precautions. To solve this problem we may have battery or generator power backup systems, or a hardware logging device that records changes to memory on a non-volatile medium. (One such device is presented in detail in [Garc83].)

Read-Only Memory. Read-Only memory (ROM) is denser and cheaper than writable memory, and hence is ideally suited for holding the static parts of a database or knowledge base. Certain types of electrically erasable ROMs that are non-volatile but have substantial write delays could also be useful in some applications. Managing the different types of memory and interfacing the static and dynamic parts of the data or knowledge base present some interesting problems.

Input/Output. A MMM may have correspondingly massive I/O needs, so facilities for moving large amounts of data to and from secondary storage are clearly desirable. For example, to load a 10 gigabyte database into memory, say after a crash, from a disk with a 2 megabyte per second transfer rate would take 84 minutes! Obviously, this MMM needs multiple disk controllers and independent paths into memory. Also needed are the operating systems facilities to partition files across multiple devices and to coordinate their transfer.

Acknowledgments. Several useful ideas and suggestions were made by Bruce Arden, Jim Gray, Andrea LaPaugh, Steve North, Ken Steiglitz, Jacobo Valdes, Peter Weinberger, and Gio Wiederhold.

REFERENCES.

[Buch78] B. G. Buchanan and E. A. Feigenbaum, "Dendral and Meta-Dendral: Their Applications Dimension," *Artificial Intelligence*, Vol. 11, Num. 1-2, 1978, pp. 5-24.

[Clar79] D. W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *IEEE Transactions on Software Engineering*, Vol. SE-5, Num. 1, January 1979, pp. 51-59.

[Cohe81] J. Cohen, "Garbage Collection of Linked Data Structures," *ACM Computing Surveys*, Vol. 13, Num. 3, September 1981, pp. 341-367.

[Comp80] Special Issue on Supersystems for the 80's, *IEEE Computer*, November 1980.

[Comp81] Special Issue on Array Processor Architecture, *IEEE Computer*, September 1981.

[Comp82] Special Issue on Highly Parallel Computing, *IEEE Computer*, January 1982.

[Cull82] R. E. Cullingford et al, "Automated Explanations as a Component of a CAD System," *IEEE Transactions on SMC*, Vol. SMC-12, Num. 2, pp. 168-182, March-April 1982.

[Cull83] R. E. Cullingford and L. J. Joseph, "A Heuristically 'Optimal' Knowledge Base Organization Technique," IFAC *Automatica*, November-December 1983.

[Evan82] D. J. Evans (Editor), *Parallel Processing Systems*, Cambridge University Press, 1982.

[Fras83] C. W. Fraser and D. H. Hanson, "A High-Level Programming and Command Language," *Proc. SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, San Francisco, June 1983.

[Frei75] W. F. Freiberger, U. Grenander, P. D. Sampson, "Patterns in Program References," *IBM Journal Research Development*, Vol. 19, No. 3, May, 1975, pp. 230-243.

[Garc83] H. Garcia-Molina, R. J. Lipton, and P. Honeyman, "A Massive Memory Database System," Technical Report 314, Department of Electrical Engineering and Computer Science, Princeton University, September 1983.

[Garc84] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A Massive Memory Machine," *IEEE Transactions on Computers*, to appear.

[Gray79] J. N. Gray, "Notes on Database Operating Systems," Advanced Course on Operating System Principles, Technical University Munich, July 1977. (Also in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham and G. Seegmuller, editors, Springer-Verlag, 1979, pp. 393-481.)

[Gray83] J. Gray, "What Difficulties Are Left in Implementing Database Systems," Invited Talk at *SIGMOD Conference*, San Jose, CA., May 1983.

[Lipt82] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, and G. Vijayan, "ALI: A Procedural Language to Describe VLSI Layouts," *Proc. Nineteenth ACM-IEEE Design Automation Conference, Las Vegas, Nevada, June 1982, pp. 467-474.*

[Mart71] W. A. Martin and R. J. Fateman, "The MACSYMA System," *Proc. ACM Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, CA., 1971, pp. 23-25.

[Shor76] E. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, American Elsevier, New York, 1976.

[Siew82] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[Siss68] S. S. Sisson, M. J. Flynn, "Addressing patterns and memory handling algorithms," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 33, Part 2, December, 1968, San Francisco, CA., pp. 957-967.

[Smit82] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September,1982, pp. 473-530.

[Spir72] J. R. Spirn, P. J. Denning, "Experiments with program locality," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 41, Part I, December, 1972, pp. 611-621.

[Wate78] D. Waterman and F. Hayes-Poth (Editors), *Pattern Directed Inference Systems*, Academic Press, New York, 1978.

[Wein83] P. Weinberger, Personal Communication.

[With83] F. G. Withington, "Winners and Losers in the Fifth Generation," *Datamation*, December 1983, pp.193-209. (These forecasts also appear in "Future Information Processing Technology, 1983," Institute for Computer Sciences and Technology of the National Bureau of Standards, August 1983.)

A MASSIVE MEMORY MACHINE

H. Garcia-Molina
R. J. Lipton
J. Valdes

TR #315

EECS Dept.
Princeton Univ.
July, 1983

A MASSIVE MEMORY MACHINE

Hector Garcia-Molina
Richard J. Lipton
Jacobo Valdes

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

ABSTRACT

This paper argues the case for a computer with massive amounts of primary storage, on the order of tens of billions of bytes. We argue that such a machine, even with a relatively slow processor, can outperform all other supercomputers on memory bound computations. This machine would be simple to program. In addition, it could lead to new and highly efficient programs which traded the available space for running time. We present a novel architecture for such a machine, and show how it can lead to reduced memory access times and higher reliability.

Index Terms: cache, computer architecture, massive memory, memory bound computation, reliability, supercomputer.

Current address of Jacobo Valdes: Imagen Corporation, 2660 Marine Way, Mountain View, Ca. 94043.

This work was partially supported by DARPA grant #N00014-82-K-0549.

December 13, 1983

A MASSIVE MEMORY MACHINE

Hector Garcia-Molina
Richard J. Lipton
Jacobo Valdes

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

1. INTRODUCTION.

This paper argues the case for a computer with a primary memory substantially larger than what is currently (or will be in the near future) available on a single machine. We do not have a specific target size for such a *massive memory machine* (MMM), but for arguments sake let us say we want on the order of *tens of billions* of bytes of main physical memory. This size is certainly larger than what any manufacturer offers today, or will probably offer in the near future. Our thesis is that such a MMM is justified, even today, by the importance of certain applications in which *memory bound computations* occur naturally. For these computations, a classic Von Neumann machine with a relatively slow (1 to 10 MIPS) processor and massive amounts of physical memory, would vastly outperform even the "supercomputers" currently being researched and would be, in addition, far easier to program.

In Section 2 we present the case for a MMM, including its economic feasibility. In Sections 3, 4 and 5 we discuss how an efficient and reliable MMM could be built.

2. THE CASE FOR A MMM.

Research efforts in the supercomputer field have tended to concentrate at the computational intensive end of the spectrum, disregarding the memory intensive applications altogether. The typical supercomputer being investigated today is a multiprocessor having up to one million processors, capable of executing up to billions of operations per second and yet have as "little" as sixty four megabytes

of physical memory [3,4,5,7].

There are many applications for which such a machine (as well as any conventional machine) would be limited by its disk to memory transfer rates. For example, consider a program which accesses a four gigabyte (4×10^9 bytes) data structure with an essentially random pattern. A machine with one hundred or less megabytes of memory can be expected to generate a page fault in just about *every* memory access, rendering its potential processing power meaningless as a measure of its performance.

More precisely, let us compare such a supercomputer with one hundred megabytes of memory and a MMM with four gigabytes of memory. Further, let us assume that the supercomputer is "infinitely fast" while the MMM runs only at one MIPS (Million Instructions per Second). Of course the supercomputer will vastly out perform the MMM on compute bound tasks. However, for the memory bound program we are discussing, assume that the supercomputer creates a page fault every f instructions , and that its disks are capable of servicing 100 requests a second. Then on this task the MMM still computes at its one MIPS rate while the supercomputer is reduced to computing at about $100f$ instructions a second. Clearly if f is small enough the MMM will be faster than the supercomputer: if f is about 100 then the speedup is 100:1! While not all tasks will cause the supercomputer to "thrash" in this way, we believe that there are a large collection of important tasks that will cause such behavior.

2.1 Applications.

An MMM will produce significant improvements for any task which references, in a relatively random fashion, a large address space. Here we will review three areas in which such tasks abound, but this list is by no means exhaustive.

(a) **Databases** [6, 22]. It is well known that many database applications are IO bound, that is, limited by the speed at which data can be transferred from disks. Clearly, if the entire database (or a substantial fraction) could reside in main memory, then the IO bottleneck would be eliminated.

Not only will existing queries be answered faster, but it will now be possible to pose new interesting queries that previously required unreasonable times to answer. Thus, users can get more useful information out of the system.

(Reliability may be a problem in a massive memory database. We will return to this and other implementation issues later.)

(b) **VLSI Design** [15]. The size of VLSI circuits being designed is growing at a fast rate. Today there are circuits with a half million transistors, and predictions of integrated circuits with as many as one hundred million transistors by the mid 90's. VLSI design tools will perforce deal with massive amounts of data, notwithstanding much cleverness in the use of hierarchical design and the encoding of information.

Many of the VLSI design algorithms have good asymptotic running times, but have very poor locality of reference. Thus, they are naturally candidates for a MMM. For example, a layout system we have designed [13] uses topological sorting for placing objects. The algorithm for sorting requires linear time, but unfortunately also requires linear space and has almost no locality. Thus, beyond a certain layout size, its actual running time is determined by the memory available: at a given point, increasing the layout size by 30% sends our computer into uncontrolled thrashing and increases the running time ten fold!

(c) **Artificial Intelligence** [16, 23]. The concept of vast data structures built mainly by the use of pointers, and hence lacking much locality of reference when accessed, brings the words "*Lisp*" and artificial intelligence (AI) to mind. Garbage collection [2] and paging times contribute substantial fractions to the total running times of many AI programs. It seems fair to say that a good fraction of AI research involves memory bound computations.

Certain AI programs, such as DENDRAL [1] or MACSYMA [14], have succinct inputs and generally produce succinct outputs, and yet may build enormous intermediate data structures. These programs are even better suited to a MMM than others. They would not even need to incur the overhead of loading the massive memory as a database or VLSI program would.

2.2 The economical feasibility of a MMM

Clearly VLSI has made computing in general cheaper. It is also clear, although not as well understood by everybody, that VLSI has made certain kinds of computing cheaper than others. One example of this differential impact involves memory and processing power: over the past few years, the price of logic circuits has decreased about 20% per year; during that same span, memory prices have decreased at twice that rate: almost 40% per year. Clearly that trend, if continued, should be very good news indeed for applications that require memory bound computations.

In fact, there are good reasons to believe that the figures given in the previous paragraph represent more than a local kink in the prices of these commodities, brought about by a vicious fight for market share in a particularly important market. Memories are the most regular integrated circuits (ICs), and thus among those which would profit immediately from higher fabrication densities. We believe that memories will be always the first circuits to profit from progress in integrated circuit manufacturing technology.

At today's prices, the cost of the ICs necessary to build a one gigabyte memory is below one million dollars. A complete computer may cost up to (roughly) 1.5 times this amount, but this is still not out of proportion with the investment necessary to equip a state of the art installations for research or production work in some of the areas identified earlier. Furthermore, if the price trends hold, the ICs necessary to build a four gigabyte memory would cost approximately 200,000 dollars by the end of the present decade.

2.3 New Programming Techniques.

A MMM is straightforward to program. Existing programs can be run on it, and if they are memory intensive, they will run very fast. However, the impact of a MMM may be even more far reaching. A MMM may alter the way we program, and this in turn may yield even grater improvements [10, 21].

For example, consider the concurrency control mechanism of a database system. Since user programs (called transactions) encounter long delays as they wait

for disk pages to be brought into main memory, the database system executes several transactions concurrently. Since the transactions are not independent (they are reading and writing the same database), their actions cannot be interleaved in arbitrary ways. The concurrency control mechanism (typically using locking) ensures that only interleavings that preserve data consistency are run. Concurrency control introduces substantial overhead and complexity into the system.

When the database system is transferred to a MMM, the disk delays disappear, and concurrency control may no longer be needed. The data required by each transaction is already in memory, so if transactions are short (as they are in many commercial systems) they can simply be scheduled sequentially. So in addition to making data available faster, a MMM may eliminate the overhead and the complexity of concurrency control.

In general, having massive amounts of memory will change our programming techniques. Data structures for secondary storage (e.g., B-trees, extendible hashing) will become obsolete. Table lookup will be practical in many more cases. For instance, instead of computing trigonometric functions with a series, we may want to have a large table of values and use simple interpolation. Digital searching [12], which improves search times at the expense of memory space, will be commonplace.

3. ARCHITECTURES FOR A MMM.

We have argued that main memory is a useful resource in many applications, and that a supercomputer with massive amounts of memory (e.g., gigabytes) is economically feasible.

But are there any technological challenges in building a MMM? Is it not just a matter of connecting all the desired memory to the chosen processor in a conventional way [17], i.e., with a very long bus? (See Figure 1.)

A conventional architecture is a reasonable one, but as we will discuss shortly there are other architectures that may be superior. The conventional architecture has two main weaknesses: memory access times and reliability.

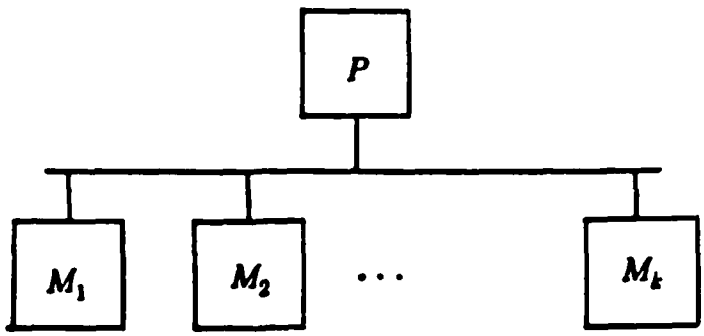


Fig. 1: *A Conventional Architecture MMM*

- **Memory access times.** Given current IC densities, a four gigabyte memory requires about one thousand devices (memory cards) on a single bus. Even with clever arrangements and higher densities, hundreds of devices per bus seem unavoidable. Building a special purpose bus to support that many devices is feasible, although not trivial. However, regardless of how the bus is implemented, as the size of the memory grows, the access times grow because of the physical distances and/or capacitance effects. At the same time, memories are becoming faster, so that the larger access times make us lose part of the advantage of having a massive memory.
- **Reliability.** As the size of the memory grows, the probability that one of its components fails also grows. A conventional architecture has no provision for graceful degradation, and hence the entire machine would be unavailable with high probability. For database applications, some type of memory redundancy is also necessary in order to avoid loss of data.

In the next sub-sections we present a new architecture which addresses the first of these weaknesses. We return to the reliability issues in Section 4.

3.1 A Novel Architecture.

Our basic premise is that the time to access memory over a long bus (i.e., one that drives hundreds of devices) is substantially larger than the access time over a short bus (i.e., one driving a single memory board). The meaning of "substantially" depends on how the buses are implemented, but for the time being let us assume that access times over a long bus are at least an order of magnitude larger than over a short bus.

A classical solution for improving access times over a long bus is to add a memory *cache* [11, 20] to the processor. (See Figure 2.) The idea is that commonly accessed data reside in the cache, and are hence available with smaller delays (both because the cache bus is shorter and because the cache memory is generally faster). Unfortunately, caching does not improve access times significantly for the programs we have in mind. A cache may be useful for holding some commonly accessed values, but as discussed in Section 2, we are concerned with programs that reference their data structures in essentially random ways. Thus, for most of the recently referenced data, the probability of being accessed next is low.

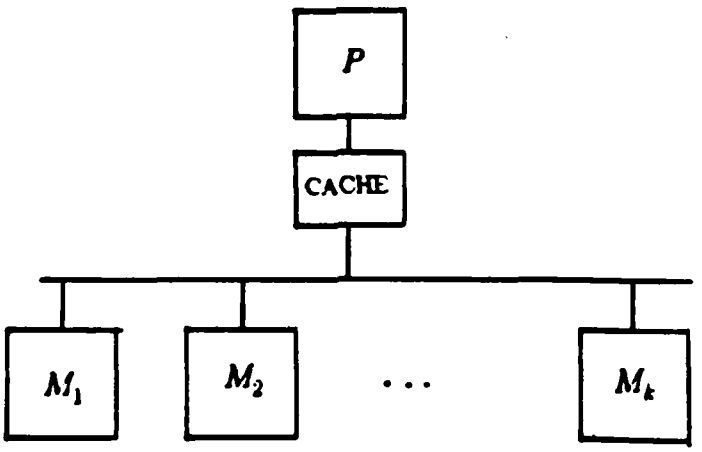


Fig. 2: *A MMM with a Cache*

If we cannot bring the data to the processor as fast as we would like, we could instead "take the processor to the data". This is precisely what the *ESP MMM* does. A schematic description of it is shown in Figure 3. (The name ESP

will be explained shortly.)

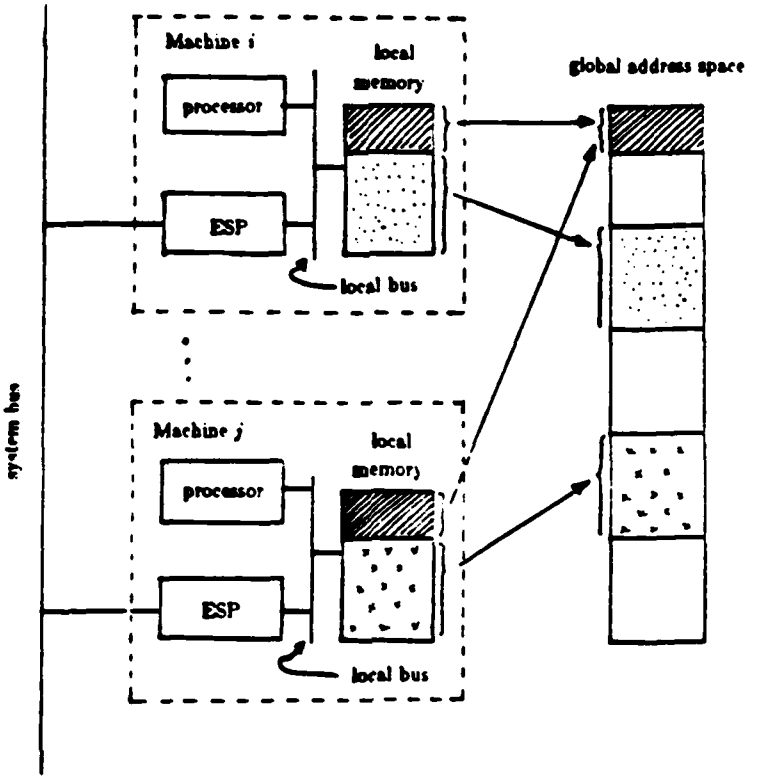


Fig. 3: *The ESP MMM*

The ESP MMM consists of a collection of standard Von-Neumann machines, interconnected by a system-wide (or global) bus that permits the broadcast of values from one machine to all the others. Each individual machine has its own processor and local memory connected via a local (short) bus. The gateway of each machine to the global bus is an *ESP* device connected both to the system bus and the local bus. (The number of machines is not critical to the architecture, but we expect a system with a few gigabytes to have a relatively small number of machines, possibly up to one hundred. This means that each individual machine has a substantial amount of memory.)

The individual processors share one same address space. This address space is distributed among the local address spaces as follows (see Figure 3). A small fraction of the global address space is replicated in each local address space; the

remainder of the system address space is covered in a non-overlapping manner by the local address spaces. An *ESP* device connected to each local bus is responsible for servicing requests that involve non-local addresses.

Even though the ESP MMM has multiple processors, it is a single instruction stream, single data stream machine (SISD) [8]. All processors execute the *same* program, which is loaded into the replicated portion of the system address space. As long as that program references locations in the shared subspace all processors will execute in lockstep and no communication through the system bus will take place. References outside the shared address space are broadcast and received on the global bus, as is illustrated by the following example.

Consider a program which references memory words w_1 through w_9. Assume that w_5, w_6, w_7 are in machine 2, and the rest of the words in machine 3. Figure 4 shows the time at which each processor receives a referenced word. In this figure we assume that fetching a word from local memory takes one time unit, and that broadcasting a word over the system bus takes two units. (We choose two units only to simplify the example. As discussed earlier, we expect the system delays to be orders of magnitude larger than the local ones.)

At time 0, all processors start; since they all run the same program, they all request word w_1. Processor 3 has w_1 locally, so one time unit later it receives it. From then on, processor 3 works at full speed, accessing words w_2, w_3, and w_4. At time 4, processor 3 requests word w_5, but since it is not local, a delay ensues.

In the meantime, the ESP at machine 3 has been broadcasting words w_1 through w_4. Word w_1 arrives at processors 1 and 2 at time 3, and the following words arrive at one unit intervals. Note that the words are "pipelined" on the bus, so that there is only *one* system bus end-to-end delay involved. Hence, after the initial delay, processors 1 and 2 start receiving and processing the words at full speed.†

During this time we say that processor 3 "has the lead", i.e., is ahead of the others. But when processor 2 references w_5, it finds this word in its local memory

† In some cases, the bandwidth of the system bus may limit the processor speed. We return to this issue is Section 3.3.

and takes the lead. The other processors must now wait until the ESP at machine 2 broadcasts w_5 and the following words. In a similar fashion, the lead changes back to processor 3 when w_8 is referenced.

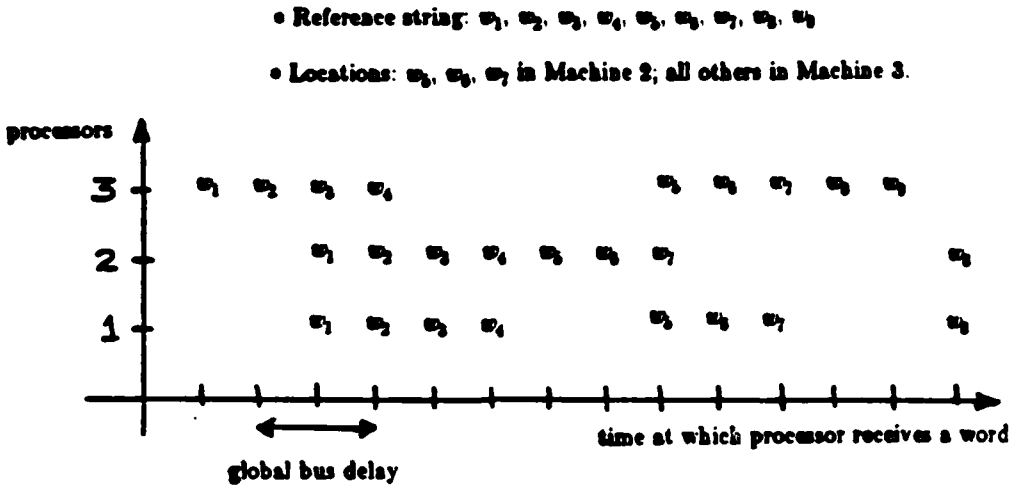


Fig. 4: *Execution in an ESP MMM*

In summary, an ESP examines each word request made by its local processor. If the address refers to the shared subspace, the ESP does nothing. If it refers to the local non-replicated memory, then the ESP reads the fetched word off the local bus and broadcasts it over the system bus. In case of a reference to remote memory, the ESP waits for the next word broadcast over the system bus, and then places it on the local bus. (This is why we picked the name "ESP" for these controllers: the remote words required appear on the system bus without having been requested, as if the controllers has ExtraSensory Perception.) In any case, the processor is not aware of the ESP controller (except for time delays); it operates as if it had a long bus linking it to all the memory units. Each local memory module must know the addresses of the data it holds, honor requests for its data, and ignore all other requests. (This is how memory modules in a conventional architecture operate.)

While the common program generates requests for data local to machine m, the processor at m takes the lead. All other processes continue execution at the same rate as m, with their ESPs supplying the data they need. These "trailing" processors, will be behind the leader by an amount of time equal to the one-way

delay time between ESPs through the system bus. When a reference to an address local to another machine occurs, that machine takes the lead.

Writes to memory can be ignored by the ESPs. When the program calls for storing into the replicated address space, all processors will execute the instruction and will update their copies. When the program modifies non-replicated storage, the processor with the data will modify it, and the rest need do nothing. (When we discuss reliability in Section 4, we will see that special precautions must be taken when writing into the non-replicated address space.)

The replicated address space is used to store the program and commonly accessed values. In addition, each processor may have registers and a cache to hold recently accessed data.

Two important things to note about the system bus are that it acts as the system "clock" and that there is no contention. The data transmitted over the bus are the timing signals that keep all processors in synchrony. (In the example of figure 4, processor 2 picks up the lead when it receives word w_4 from processor 3.) Since non-replicated data is found only at a single machine, only one ESP will ever broadcast at a time. This means that the bus protocols will be very simple, and hence transmissions can be fast.

The ESP architecture has the following advantages over a conventional one:

(1) The local machines have conventional architectures. They may be used independently when the MMM is not needed.

(2) For fully random references, memory access times are cut by roughly a *factor of two*. In a conventional machine, the address must be transmitted on the system bus and the referenced datum must be transmitted back. In an ESP machine, no addresses have to be transmitted on the global bus: each datum appears on the system bus without having been requested. That is, since references are random, each memory access will cause a lead change. But these lead changes only involve a one-way broadcast, and thus, half the delay encountered in a conventional architecture.

(3) The ESP MMM will reward "locality of reference" by minimizing "lead changes" in programs that exhibit it. That is, if two or more references fall within the same memory module, then the access times are reduced to local bus times. The fewer the lead changes, the faster the ESP MMM will execute.

Locality in this context, however, has a wider meaning than in a conventional memory cache or virtual storage system. Here, locality *of reference* means that two references are local to the lead machine, and this machine may have a substantial chunk of memory (probably tens of megabytes). In the next sub-section we will explore these issue in more detail.

What is the price we pay for these advantages? Obviously, we have replicated processors and some data. Given current pricing trends, the cost of this extra hardware should be reasonable, at least compared to the cost of the massive memory. (The processors do not have to be high performance ones. Recall that for the applications we have in mind, the limiting factor is the speed at which data can be retrieved from memory, and not the speed at which the processor *manipulates* it.)

What we have *not* sacrificed is simplicity and ease of programming. The processors and memory modules are conventional. The ESP architecture is transparent to the user program. The task of distributing the global address space to the spaces of the individual machines can be relegated to a sophisticated loader.

3.2 Program Locality.

The potential performance improvements of an ESP MMM over one with a conventional architecture hinge on two main factors:

(i) The "locality" exhibited by the program, and

(ii) The memory access times over the system and local busses.

In this sub-section we study the first factor in more detail. The bus times are discussed in the following sub-section.

The ESP MMM utilizes several mechanisms to improve memory access

times: (1) registers and caches at each processor to hold recently accessed values; (2) a replicated address space to hold the program and commonly accessed values; and (3) the ESP mechanism, which lets the leading or controlling processor move to the memory module where the data resides. The first two mechanisms can be easily incorporated into a conventional MMM, so the decisive factor is clearly the ESP mechanism.

What does the ESP mechanism give us that the others do not? In order to answer this question, let us postulate a simple *data* reference pattern. (We are not interested in the *instruction* reference pattern, since the entire program is replicated in all machines.)

Suppose that the M memory words of the MMM are divided into *blocks* of B words each. A block is the unit of data transfer between the memory and a cache. We assume that the location of the next referenced block depends only on the location of the most recently accessed one. Specifically, Figure 5 gives the probability distribution of the next reference. There is a set of a blocks, centered on the last referenced block, that have a high probability p of being accessed next. All other blocks have a much lower probability q. (For simplicity, we assume that when the last reference is within $a/2$ blocks of the ends of the memory, the distribution wraps around.) We assume that a is odd.

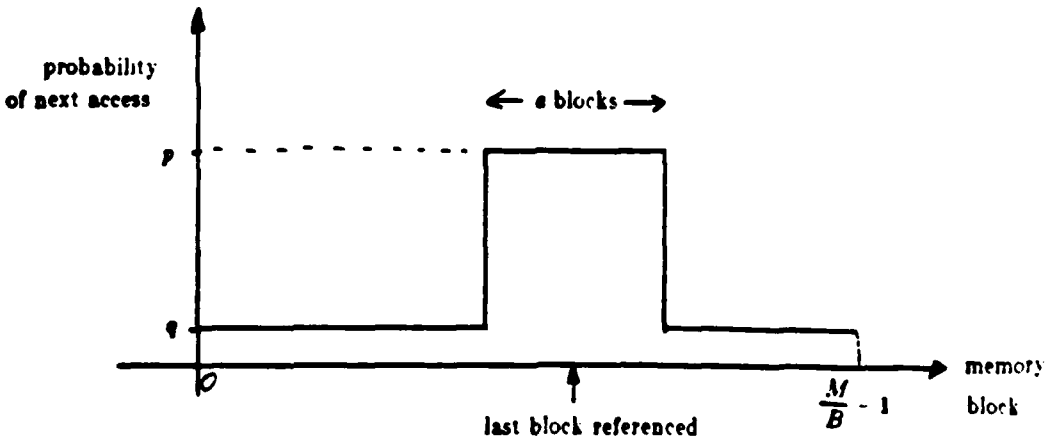


Fig. 5: *The Probability Distribution.*

Our experience tells us that this is, in an idealized way, the way programs reference their data (e.g., see [19, 20]). For example, consider a program that simulates a VLSI chip. When a transistor is referenced, several contiguous words may be referenced. The next transistor reference is likely to be to a connected one, and if the circuit is represented in a reasonable way, it will be close to the previous one. Here "close" may mean within a few thousand bytes, so our high probability window, a, may be relatively large.

The parameters a and p define the *locality* of the program. As a shrinks and/or p grows, the program exhibits more locality, and as a grows and/or p approaches q, the references become more random (i.e., the distribution becomes flatter).

Note that this distribution ignores other types of data locality that may also be exhibited by programs. For instance, programs may have time locality (i.e., tend to reference recently accessed data) or may access certain fixed locations with high probability. Since these types of localities are exploited by data caches, the distribution we have selected to study will highlight the strengths of the ESP mechanism, not of caches. This is precisely what we want to do.

Using this probability distribution, we have analyzed the performance of an ESP mechanism (where processors have no registers or caches) and of a simple cache. The analysis is described in [9]. Figure 6 presents some typical results. The figure shows the hit ratio for the cache (h_C) and the ESP mechanism (h_E), as a function of a, the high probability window. For the cache, the hit ratio is the probability that the next referenced word is in the cache. For the ESP, it is the probability that the next word falls in the same machine as the previous word. (In the figure, locality decreases from left to right.)

If on each memory reference the cache can fetch a significant portion of the "high probability of next access" window, then the cache performs very well. (That is, if a is close to 1 block.) In this case, either the program has very high locality or the system bus feeding the cache is very wide. In this case the ESP does not have any advantages over the cache.

At the other extreme (very large a), references are fully random and both mechanisms have a hit ratio of 0. In this range, the ESP is superior by roughly a

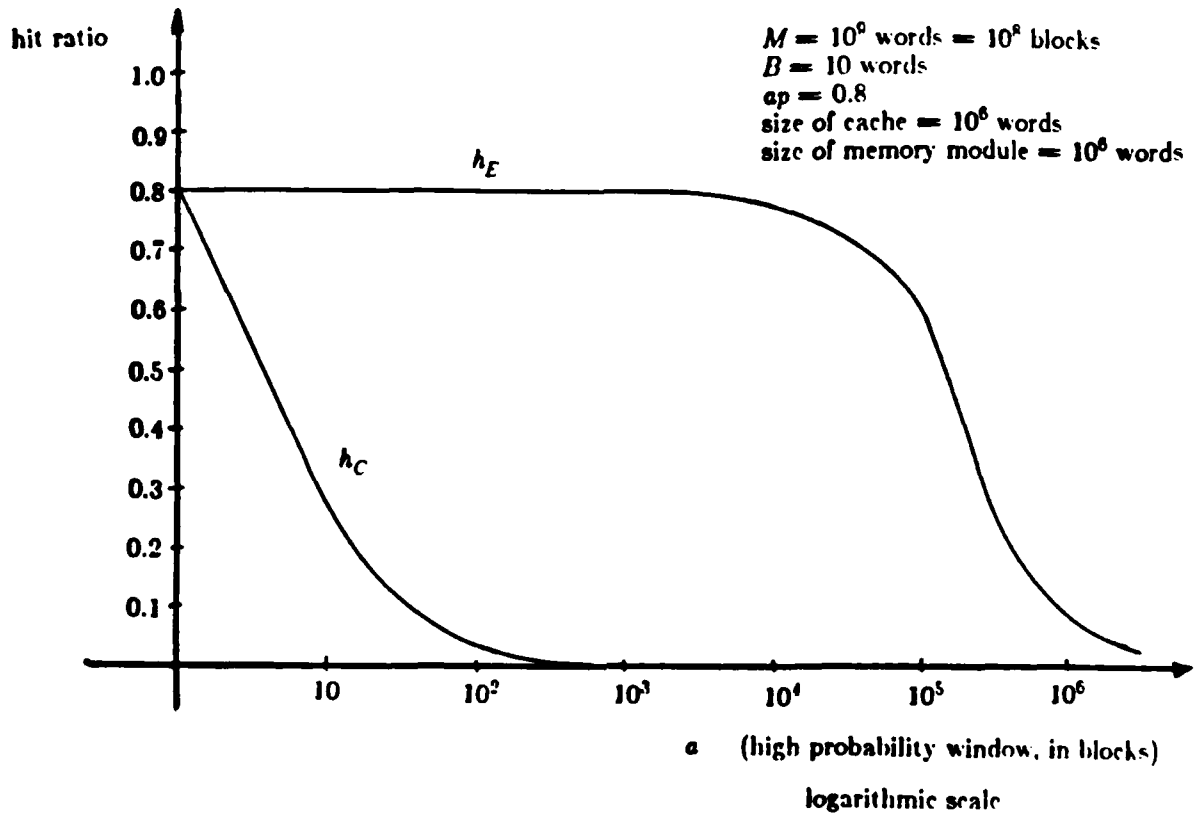


Fig. 6: *Hit Ratios for ESP and Cache*

factor of two because, as we discussed earlier, addresses need not be broadcast.

In between is a large range of localities where the ESP performs substantially better than the cache (from a equal to 4 or 5 until a is roughly the number of blocks in a memory module of the ESP.) In this area, most references using the ESP mechanism are local. On the other hand, with a cache, most references continue to rely on the system bus. This is because the cache mechanism retrieves data from memory in very small units, on the order of a few words. The improvement will be, roughly, the ratio of system bus access times to local bus times.

The programs that will use a MMM, as we argued in Section 2, are memory intensive ones, programs that cause a virtual memory system to thrash. Thus we expect these programs to operate in the range of localities where the ESP mechanism does pay off.

([9] presents more results, and also considers other probability distributions. The trends obtained are similar to what we have presented here.)

3.3 System and Local Bus Access Times.

The performance improvements of an ESP MMM over a conventional architecture depend on the value of the system bus access time, D, and the local bus time, d. So far we have assumed that D is much larger than the cycle time of the processor(s) and than d. If these assumptions do not hold, then the gains of the ESP mechanism will be limited. For example, if we can implement a system bus with D small compared to the cycle time of the processor, then obviously, cuting the bus access times by half is not important.

We have also assumed that the system bus has sufficient bandwidth to pipeline data as fast as it is fetched from a local bus. If this is not the case, the local bus will have to be slowed down, effectively increasing the value of d.

The values of d and D depend on the hardware used to implement the MMM, as well as on the size of the memory, and therefore, it is difficult to reach any definitive conclusions. For example, the delay D is a function of the bus physical distance and the number of loads. These parameters are in turn a function of the memory size and the packaging density. The bandwidth of the system bus may be limited by skew on its lines, which in turn is a function of the bus length. Of course, the bandwidth will depend on the technology used, e.g., an optical bus will have much higher bandwidth than a conventional one.

In summary, it is not possible to state whether the ESP mechanism is advantageous unless most hardware parameters are known. However, we can discuss two *general* implementation scenarios where certainly D is significant as compared to the cycle time, and where d is orders of magnitude less than D. In both of these cases, the ESP MMM performs very well.

- **Processor and Memory on a Chip**. It will soon be possible to build a reasonable processor with a few megabytes of memory, all on a single VLSI chip. These chips will be ideally suited for the construction of an ESP MMM. The time to access on-chip memory (d) will be very small, since

small currents and small distances are involved.

The limiting factor in this implementation will be the rate at which ESPs can broadcast data out of the chip, into the system bus. However, an optical bus may provide the necessary throughput.

- **Sharing Memory on Existing Computers.** Suppose that we already have an installation with several computers (maybe 2 or 3, maybe 100 or 200) connected via a local area network. The ESP architecture gives us a way to combine these resources into a single MMM, when it is needed. Clearly, local memory access times are significantly less than transmission times over the network, so the ESP is a useful idea Each existing machine would be provided with an ESP controller, and the network protocols (for MMM operation) would be simplified, e.g., there is no contention, no need for packet headers. (This assumes that while the machines operate as a MMM, the network has no other users.) A program requiring more memory than is available at a single machine (even if it only needs the memory of 3 or 4 other machines) can be sped up considerably. There will be improvements even if its references are totally random, since page faults (with seek, rotational, and substantial data transfer delays) will be replaced by fast (and probably short) network messages.

  For some programs it may be possible to implement the ESP mechanism fully in software. If a program has a distribution similar to the one of the previous sub-section, and if a is less than the memory at each computer, then lead changes will be infrequent. A lead change can then be implemented by sending a message with the state (e.g., contents of registers) of the lead machine to the next leader.

4. RELIABILITY ISSUES.

In this section we briefly outline how a MMM can be made reliable. For concreteness, we consider an ESP MMM. Some of the techniques we discuss can also be used on a conventional architecture MMM, although the ones that exploit the replicated processors obviously cannot.

The first step is to make the individual components of the MMM reliable. For this, either very reliable hardware can be used, or error detection (and correction) code bits can be added to the memory modules, buses, and processor registers. (Deciding how many bits and where they should be located is not a trivial problem, but solutions exist and are well understood [18].) In spite of this protection, component malfunctions can occur, and the system as a whole must cope with them. In the rest of this section, we discuss possible strategies for this.

We distinguish two types of undesirable events. *Errors* are transient malfunctions of a component. If an error occurs during an operation (e.g., broadcasting a value on the system bus, or reading a value from a memory module), then the operation can be repeated and with high probability it will be correct. On the other hand, *failures* are longer lived malfunctions that can be remedied only with outside intervention.

Throughout our discussion, we will assume that errors and failures can be detected immediately, say by error detecting codes. When a malfunction is detected, the hardware will automatically retry the operation. After a number of unsuccessful attempts, the malfunction is declared a failure, and no further operations are executed by the failed component.

To coordinate recovery actions, we introduce a *master processor* in the ESP MMM. (See Figure 7.) It is simplest to make this a specialized processor which does not run user programs (although this is not necessary). The master has two buses linking it to the other machines. The *secondary system bus* connects the master directly to the memory modules (using dual input ports†), forming a conventional architecture. This bus is used to access memory in case a local bus or ESP controller fails. It can also be used to run the system as a conventional MMM when the main system bus fails. The *control bus* links the master to all other processors. Through this bus the master can observe the status of the processors, and can issue recovery commands.

---

† Dual port memory is currently very expensive, but for no inherent reason VLSI should change this situation.

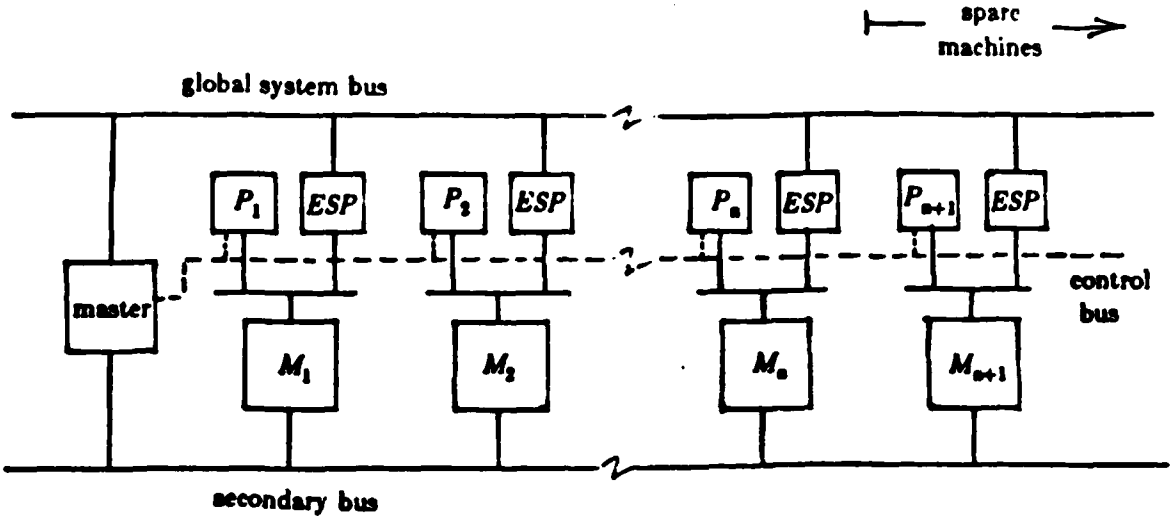


Fig. 7: *A Reliable MMM*

4.1 Coping with Errors.

The tight synchronization of the processors complicates the recovery from errors. If the leading processor detects an error, then there is no problem. It can retry the operation, and the rest of the processors will be delayed accordingly. However, if a trailing processor detects an error and attempts to repeat the operation, it will get out of synchrony. If the operation was the reception of a word from the system bus, then the processor cannot even repeat the operation since it does not control the bus.

The solution is to simply halt any trailing processor that detects an error. The rest of the system will continue operating normally until control passes to a halted processor. At this point the master detects the problem (e.g., it sees no activity on the system bus), and directs one of the active processors to broadcast all of its state information (i.e., contents of registers, cache, and replicated address space) over the system bus. The halted processor(s) loads the information and is then ready to go.

In Section 3.1, when we discussed memory write instructions, we stated that writes could be ignored by the ESP controllers and by the processors that did not have the data being updated. However, in a reliable system, this is no longer the

case. To illustrate, consider a memory reference string r_1, r_2, w_3, r_4, r_5, where r_1, r_2, r_4, and r_5 are read references to data in memory module A, and w_3 is a write into module B. For the first two read references, the processor at A takes the lead. When this processor encounters w_3, it cannot assume that B will execute the write because B could be out-of-synchrony. Thus, A must receive an acknowledgment from B indicating that w_3 was successfully performed before taking the lead again to perform r_4 and r_5.

To solve the problem, we handle writes to non-replicated data with the same protocol that is used in reading. In our example, when A encounters w_3, it waits for an acknowledgment from its ESP, just as it would wait for data had the instruction been a read. At this point, the lead passes to B which executes w_3, and its ESP broadcasts an acknowledgment, just as data would be broadcast had w_3 been a read. If B is out-of-synchrony, the system halts when the lead passes to it (at w_3), and the recovery starts.

4.2 Coping with Failures.

Processor, ESP or Local Bus Failures. When a processor, its ESP, or its local bus fails, we again wait until control is passed to it. The master must then allocate the functions of the failed machine to a spare machine. For this, it must be possible to dynamically redefine the address space managed by a machine. Each memory module and ESP controller would have registers with this information, and the registers would be loadable by the master (through the control bus). Once the new machine is allocated, the master copies the non-replicated data of the failed machine into the new machine (using the secondary bus). Finally, the state information is broadcast (as if an error had occurred), and processing continues.

Memory Failures. When a memory module fails, we cannot simply allocate a new machine to replace it. Each module contains data that are not available elsewhere. One strategy for dealing with these failures is to abort the program. The failed machine is replaced (as discussed above) and the program is re-started

from scratch.

If abortion is undesired, then storage must be replicated. Furthermore, for some applications (e.g., databases) at least one of the copies must be non-volatile. The simplest solution is to fully duplicate each memory module (within each machine), and to make one of them non-volatile by supplying backup power from batteries (and possibly a generator). However, the following solution may be less expensive.

Notice that (normal) reads can be handled by the primary copy, so only writes must be performed efficiently by the secondary copy. (A program writing a word must wait until both copies are safely written out, and hence *both* writes must be done fast.) Therefore, when a processor issues a write command, a copy of the address and new value are stored in a small, temporary buffer. (The buffer can be made non-volatile with battery backup power.) After this operation (and the write to the main copy) the processor may proceed with its work. Meanwhile, a separate controller executes the updates recorded in the buffer onto a copy kept on a slow, non-volatile device like a disk or a drum. This device can be shared among several machines to reduce costs.

When a backup copy is needed, it is loaded up into a functioning primary memory module. Any unexecuted writes in the buffer are "played back" to obtain an up-to-date-copy.

The buffer shields the slow device from bursts of write commands, but if the average rate is too high, the buffer will fill up and the processor will have to wait. In this case, the copy on the slow device can also be kept in journal form, i.e., as address, value pairs. In this form the writes can be stored sequentially, avoiding seek and rotational delays. Of course, when the secondary copy is needed the recovery will be slower since the writes must be played back. To decrease these times, periodic dumps of the primary memory should be taken. With some care, these dumps can be stored in the same slow device, concurrently with the journaling of the writes, and in the same sequential file.

Power Failures. Loss of power to the entire MMM is equivalent to a simultaneous failure of all the machines. When power is restored, the contents of each

memory module can be recovered using the techniques described above, but the state of the processors cannot be restored since it was lost everywhere. Thus, a power failure affects the MMM as it does all other computers. To avoid restarting a task(s) from scratch, each task should periodically save its state in memory (called checkpointing), and at recovery, a task can be restarted at its latest checkpoint.

5. EXPLOITING PARALLELISM IN THE MMM.

Although the MMM with ESP controllers is a single instruction, single data stream machine (SISD), there is no reason why the multiple processors cannot be used individually when needed. What is more, with some simple extra hardware, the MMM can be made into a reconfigurable massive memory *and* parallel processing machine. This machine could switch, on demand, from SISD to SIMD (multiple data streams) or MIMD (multiple instruction and data streams), and back. Clearly, this machine would be harder to program than a plain MMM, but it could yield significant performance gains on programs that could be decomposed into parallel components.

There are several strategies for reconfiguring the MMM. For example, suppose we want to perform an associative search through a large table, and we know there is exactly one matching entry. If we split the table across the MMM memory modules, each processor could search in parallel. For this, we can add a special associative search instruction. When a processor encounters it, it forks and initiates a local search. Before the search, the processor informs the ESP so it will not broadcast the data fetched from local memory. When the processor finds the entry, it notifies the ESP, which in turn broadcasts it, just as if it had been data fetched from memory. When other ESP controllers receive the broadcast, they interrupt their processors (the entry has been found elsewhere), and hand them the result.

After the associative search, the processors continue (in SISD mode) as they would after any other instruction. The result of the search synchronizes the processors, just like data fetched from memory does for other instructions.

If the search may yield multiple matches, or if we want each processor to compute a value, we need a different mechanism for joining the instruction streams into one. One alternative is to use a special hardware device to detect the termination of the forked processes. As each processor finds or finishes computing its value, it stores it into its local, non-replicated address space, notifies the special device, and waits. When all finish, the device signals the processors to continue in SISD mode. Note that all the results are available to the SISD MMM, since they are stored in the memory it can access. A second alternative, processors could fork for a pre-determined amount of time or number of cycles. Under normal circumstances, this time should let all processors complete.

The fork and join operations we have described are limited. Only the MMM can spawn processors, and it cannot continue until they have all joined back. Of course, we could design more general control structures, but as they become more flexible, they become harder and harder to program. We feel that the mechanisms we have described strike a good balance between complexity and the performance that might be gained.

6. CONCLUDING REMARKS.

If we look at the ratio of memory size to processor speed of past and present commercial computers, we find that most are within an order of magnitude of one megabyte per MIPS. (The value one megabyte per MIPS is called "Amdahl's constant".) All supercomputers in use, and most of those being developed, have ratios well below this value, and are targeted for computationally intensive problems. The machine we proposed here, on the other hand, would have a memory to speed ratio of 100, 1000 or more. We have argued that such a machine would speed up memory bound programs like no other computer could. We also asserted that a massive memory machine having unconventional architecture and features would be more efficient and reliable. Yet, in spite of its novel structure, this machine would be simple to program.

We have only sketched the main features of a massive memory machine and the ESP architecture, but of course, there are many other important issues that must be resolved before such a machine can become a reality. In concluding, we

mention some of these issues:

Input/Output. A MMM will have massive input/output needs. The multiple processors of the ESP MMM are helpful here, for each one can have its own secondary storage device. This way, data can be loaded in and out in parallel. As an alternative, groups of processors can share their IO devices. Management of files is complicated, though, since files will be partitioned across several devices.

Virtual Storage. No matter how large the massive physical memory is, there will always be some programs that require more. A virtual storage system is the obvious solution, but there are several ways in which the virtual store could be mapped into the physical memory. The number of processors having virtual memory facilities can also be varied.

Programming Language. It may be useful to have language constructs for specifying the data to be placed in the replicated address space, and for indicating what data should be placed in the same memory module of the ESP MMM. A smart compiler could also automate some of these decisions.

Special Processors. The ESP MMM can operate with conventional processors, but this does not mean that they *must* be conventional. Some processors could have extra hardware for complex, less frequent operations (e.g., floating point arithmetic). The results of these operations would be broadcast to the rest of the processors. This strategy would reduce the cost of the majority of processors.

Also, special instructions could be added to improve the efficiency of the ESP mechanism. For example, using a conventional instruction set, a block data transfer between two modules will incur two lead changes for each word transferred. However, a special block transfer instruction could make the source processor transmit the entire block (without waiting for acknowledgments for the writes) and the destination processor store the block. At the end of the transfer, the destination ESP broadcasts an acknowledgment, and this is interpreted by the other processors as the end of the special instruction. With this instruction, the number of lead changes is reduced to two, regardless of the size of the block. Additional instructions could improve other operations.

Acknowledgments. Several useful ideas and suggestions were made by Bruce Arden, Richard Cullingford Jim Gray, Peter Honeyman, Andrea LaPaugh, Steve North, Ken Steiglitz, Peter Weinberger, and Gio Wiederhold.

REFERENCES.

[1] B. G. Buchanan and E. A. Feigenbaum, "Dendral and Meta-Dendral: Their Applications Dimension", *Artificial Intelligence*, Vol. 11, Num. 1-2, 1978, pp. 5-24.

[2] J. Cohen, "Garbage Collection of Linked Data Structures", *ACM Computing Surveys*, Vol. 13, Num. 3, September 1981, pp. 341-367.

[3] Special Issue on Supersystems for the 80's, *IEEE Computer*, November 1980.

[4] Special Issue on Array Processor Architecture, *IEEE Computer*, September 1981.

[5] Special Issue on Highly Parallel Computing, *IEEE Computer*, January 1982.

[6] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1981.

[7] D. J. Evans (Editor), *Parallel Processing Systems*, Cambridge University Press, 1982.

[8] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, September 1972, pp. 948-960.

[9] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "Analysis of the Massive Memory Architectures", Technical Report 313, Department of Electrical Engineering and Computer Science, Princeton University, May 1983.

[10] J. Gray, "What Difficulties Are Left in Implementing Database Systems", Invited Talk at *SIGMOD Conference*, San Jose, CA., May 1983.

[11] K. R. Kaplan, R. O. Winder, "Cache-based Computer System," *IEEE Computer*, March, 1973, pp.30-36.

[12] D. E. Knuth, *The Art of Computer Programming; Volume 3: Sorting and Searching*, Addison-Wesley, 1973.

[13] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, and G. Vijayan, "ALI: A Procedural Language to Describe VLSI Layouts", *Proc. Nineteenth ACM-IEEE Design Automation Conference, Las Vegas, Nevada, June 1982, pp. 467-474.*

[14] W. A. Martin and R. J. Fateman, "The MACSYMA System", *Proc. ACM Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, CA., 1971, pp. 23-25.

[15] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[16] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.

[17] A. V. Pohm, O. P. Agrawal, *High-Speed Memory Systems*, 1983.

[18] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[19] S. S. Sisson, M. J. Flynn, "Addressing patterns and memory handling algorithms," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 33, Part 2, December, 1968, San Francisco, CA., pp. 957-967.

[20] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September,1982, pp. 473-530.

[21] P. Weinberger, Personal Communication.

[22] G. Wiederhold, *Database Design*, McGraw-Hill, 1977.

[23] P. H. Winston, *Artificial Intelligence*, Addison-Wesley, 1977.

New approach for a constraint-based layout system

José M. Mata

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

Abstract

Procedural languages have been used successfully to describe VLSI layouts, especially those languages that are constraint-based, since their user doesn't have to worry about absolute positions in the layout. Some complaints about these languages are difficulty of use, efficiency in handling large layouts, area efficiency, need for design rule checker, and difficulty to implement and modify.

We present a proposal for a simple and powerful constraint-based layout system, that can be used as part of a larger CAD system for VLSI. Our approach consists basically in a structured way to describe the layout, use of an intermediate form to represent the layout (in terms of cells, not boxes), and a hierarchical constraint solver that deals with large layouts.

1. Introduction

There are many advantages of using a procedural language to describe a VLSI layout. Many of the existing languages [2,6,7] are constraint-based, that is, the program that describes the layout generates a set of linear equations whose solution gives the final layout.

The main issues addressed by constraint-based layout languages are:

- regarding VLSI design as a **programming task**, as opposed to a geometric editing task. This, among other advantages, makes the design easier, facilitates the division of labor, allows parametric design, facilitates the update of layouts, and gives a good documentation of the design.
- creation of an **open-ended tool**. Graphics editors tend to be closed tools, in the sense that it is hard to automate the process beyond what the original design of the system allowed. With a procedural language, a program can have some processing (for the description of a router, for example), or the program can be generated automatically with other tool; in this case, the layout system can be part of a more general CAD system.
- creation of tools that are **simple to use**. Reducing the number of information that the user has to give in order to describe the layout simplifies the design; the user should describe the layout conceptually, leaving the determination of absolute sizes or positions to the system. Also, allowing hierarchical design makes the design easier. Another aspect is error detection: the system should help the user to find errors in the design

- eliminating the need for **design rule checking**. From the description of the layout components and topology, the system generates constraints taking into account the design rules of the fabrication process.
- efficient **node extraction** for simulation. The nodes can be extracted from the program, that describes the circuit conceptually, instead of being extracted from the final layout.

Different layout systems address these issues in different ways, each system having its drawbacks. In the next session we give an overview of a constraint-based layout system, ALI2, and then we present our proposal for a new system.

2. A layout system: ALI2

The ALI2 layout system [4,5,7] was developed at Princeton, and has been used successfully, with many VLSI designs completed and the chips fabricated.

ALI2 is a superset of Pascal. The objects manipulated by an ALI2 program are *cells* and *wires*, besides the usual Pascal objects. There are statements to instantiate a cell with a given orientation, and to specify the relative position of cells.

Basically, in an ALI2 program the user describes the objects (cells and wires), and their topological relations; no absolute coordinate is ever mentioned. The execution of the program generates constraints of the form $x_i = x_j$ and $x_i - x_j \geq d$ (x and d integer, $d > 0$), taking into account the design rules of the fabrication process (NMOS). The constraints for the X and Y coordinates are independent. These constraints are solved in linear time using the union-find algorithm (linear for pratical purposes) and the topological sort, giving the final layout in CIF (Caltech Intermediate Form).

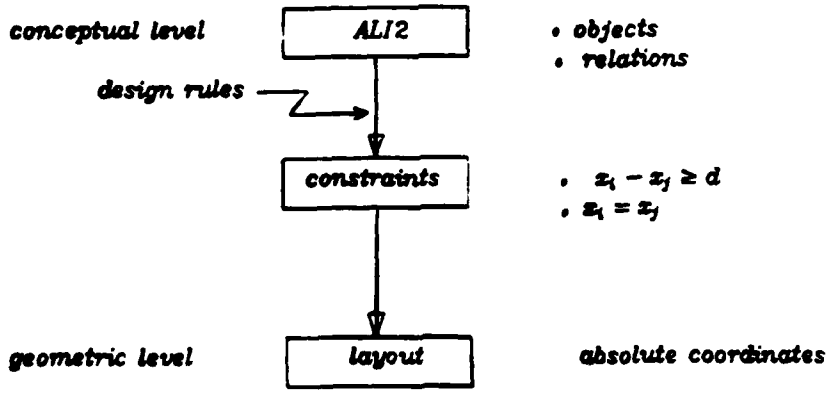


Fig. 1 - Layout generation process

An example of an ALI2 program follows, with the corresponding layout.

```
chip shiftregister (output);
wiretype
   polywire = wire (poly, 2*lambda, nullsignal);
   diffwire = wire (diff, 2*lambda, nullsignal);
   metalwire= wire (metal,4*lambda, nullsignal);
   fivewires (lr: layer) = bus
                              w1: polywire;
                              w2: metalwire;
                              w3: wire (lr, minwidth(lr), nullsignal);
                              w4: metalwire;
                              w5: polywire;
                           end;
   wirevar ll, rr: fivewires (poly);
   cell shift ( left ll: fivewires; right rr: fivewires ); rigid ( 'shift.rc' );
   cell shiftregister ( left inbus: fivewires; right outbus: fivewires )
      .          (length: integer);
      wirevar temp: fivewires (poly);
      begin
         if length = 1
            then create shift ( inbus, outbus )
            else begin
               create shift ( inbus, temp );
               create shiftregister ( temp, outbus) ( length - 1 )
         end {if}
      end;
   create shiftregister ( ll, rr ) ( 3 )
   end.
```

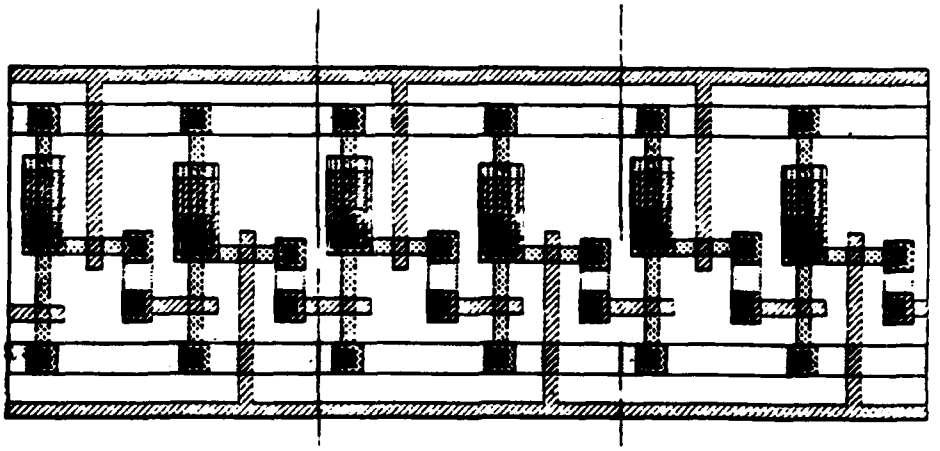


Fig. 2 - ALI2 program and layout

A cell in ALI2 can be flexible (its size will depend on the context in which it is instantiated), or rigid (fixed size, like pads or other cells generated previously by ALI2 or by other tools). The complete ALI2 system includes a switch-level simulator, a PLA generator, and some programs to interface with CIF code.

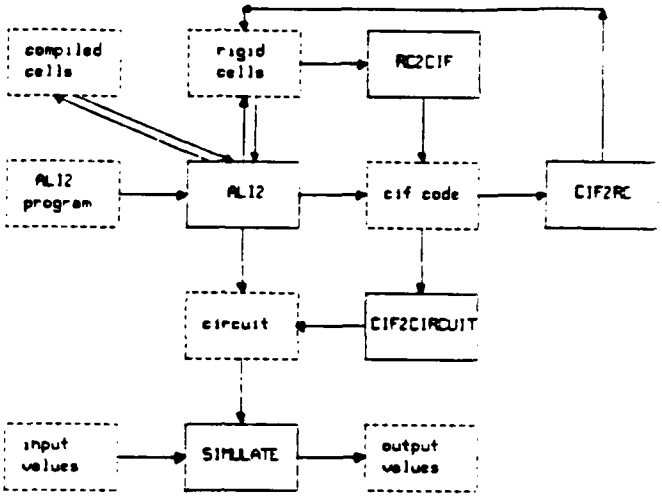


Fig. 3 - The ALI2 system

One of the chips designed using ALI2 was a n-bit parallel adder, taken from [8]. n is a parameter in the program. The 8-bit adder was fabricated, and the chip works according to the specifications.

Although all the issues mentioned in section 1 are addressed in ALI2, there are still some minor problems:

- **ease of use**. The main complaint about ALI2 is having to name wires; depending on the regularity of the layout we may have too many wires. Other problem has to do with rigid cells (fixed size): as they involve a constraint of the form $x_i - x_j = d$ $(d > 0)$ (that is replaced by $x_i - x_j \geq d$ and checked after solving), if we don't use rigid cells correctly we may get a system of constraints with no solution.
- **hierarchical design**. Although the program in ALI2 can be hierarchically structured, the constraint solver is not hierarchical, bringing space problems when the number of constraints is large. If we create solving hierarchy by using rigid cells, that brings the problem with rigid cells mentioned above.
- **implementation considerations**. Changes in the system are difficult to make, since everything is related: language aspects, design rules, and constraint generation.
- **design rule checking**. In an ALI2 program it is possible to leave two cells without any explicit or implicit topological relationship, thus causing some design rule violation. So, the layout produced by an ALI2 program is not guaranteed to be free of design rule violations.

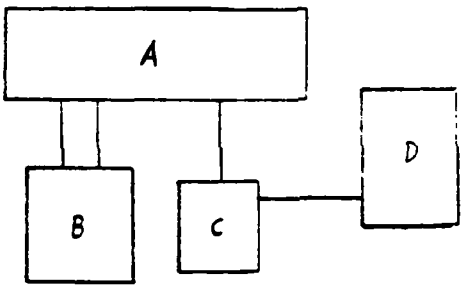
3. Our approach

We would like to have a constraint-based layout system with the following features:
- easy to use;
- powerful;
- efficient;
- easy to implement;
- hierarchical;
- guaranteed to produce layouts with no design rule violations;
- interface with high-level languages, graphics editors, simulator, and CIF code.

Our approach consists basically of 3 ideas: imposing a structured way to describe the layout, use of an intermediate language, and use of a hierarchical solver.

3.1. Structured layout

Our first basic idea is to impose a structured way to describe the layout, and take advantage of this structure. The basic unit in the layout is still a cell, that corresponds to a rectangle, and cells are composed using unary operators that specify orientation (rotation or flipping) or binary operators (*left*, *right*, *above*, *below*).



(A above (B left C)) left (rotated90 D)

Fig. 4 - Structured layout description

A cell corresponds to a bounding box (rectangle), possibly with wires on each side. There are 5 kinds of cells:
- *system*: transistor and contact;
- *named*: cell previously defined in the program;
- *external*: library cell;
- *rigid*: cell with fixed size;
- *local*: cell created by composition of two other cells.

Now the context where a cell is instantiated is well defined. Only for system cells we have to specify the parameter wires (layer and width); for other cells the wires are defined implicitly. For example, if *C* and *D* have the structure shown below, the composition *C left D* produces the right constraints relating *C* to *D* and relating the wires connecting them, and leaves a cell with the structure shown.

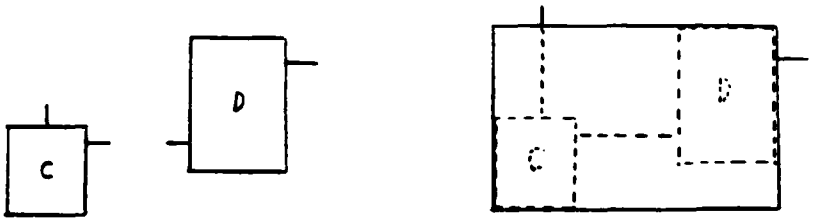


Fig. 5 - Composition of cells

There are many advantages of using this structured approach:

- no wire naming is necessary; at any point the system knows which wires are hanging up, and will use the proper wires when instantiating the next cell;
- the layout can be described in a compact way;
- separation is directly obtained, and the layout is guaranteed to be free of design rule violations;
- since the context in which a cell is instantiated is well known, there is room for local optimization, like cell separation;
- easy implementation: each cell has some semantic information associated with it; composition of cells means combination of this semantic information. If we use a grammar to describe this layout language, the construction of the layout can be done when parsing (bottom-up). In fact, this is similar to the way the system for typesetting mathematics EQN [1] was designed and implemented. In EQN, equations are pictured as a set of "boxes", pieced together in various ways.
- easy error detection: if a cell is instantiated in the wrong context, is is easy to point out exactly where the error occurred; the only possible error when solving the constraints is when a rigid cell gets stretched, and that is also easy to point out.

At first, it seems to the user that having to describe the layout in such structured fashion is a strong restriction, and may cost a lot of chip area. Our experience shows that this is not the case. First, the ordered structure comes naturally when we go from the floor plan to the program. Second, our choice of the floor plan is what is going to affect the cost in area.

3.2. Intermediate language

The idea is to separate language aspects from layout aspects, or user aspects from system aspects. For layout aspects or system aspects we mean generating cells and wires (that is, constraints), according to the design rules. For language or user aspects we mean the high level language used to describe the layout, and its implementation; here, language also applies to graphics language.

We can define an intermediate language to describe the layout, as explained in the previous section. Considering that there are no generic cells at this level (no conditional processing, no recursion, ...), it is possible to have a compact form for the layout description. This form can be automatically generated from a high level language, from a graphics editor, or even generated by hand.

Now, the high level language available to the user can be as sophisticated as we want, provided it generates code in this intermediate form. As it should be clear now, the implementation of the intermediate language and of the high level language are independent.

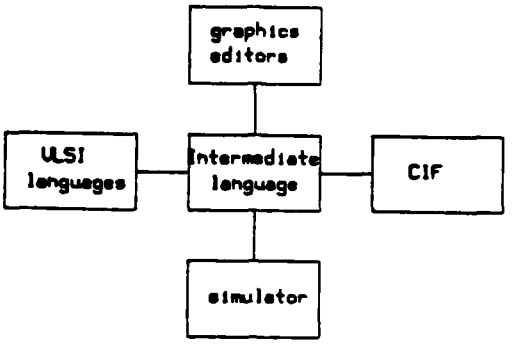


Fig. 6 - The role of the intermediate language

3.3. Hierarchical solver

The way to specify large layouts keeping the number of constraints within limits is to use hierarchy: solve the constraints for subcells and make then rigid cells (rectangles of fixed size, with the position of the inside elements already determined). Rigid cells involve constraints of the form $x_i - x_j = d$ $(d > 0)$. We developed an algorithm [3] that solves m constraints of the form $x_i - x_j \geq d$ $(d > 0)$ and n constraints $x_i - x_j = d$ $(d > 0)$ with time complexity $O(m.n)$. As n is usually small, the algorithm is pratically linear on m, and is very efficient. It is implemented in the new ALI2 system, giving excellent results.

4. Conclusions

We are now in the process of completing the design of the system. A lot of experience was gained with the ALI1 and ALI2 systems. Only minor details of the syntax of the intermediate language are left. For the high level language, our first implementation will be a set of procedures to be used in Pascal or C, that allows the user to define and instantiate cells. Generic and recursive cells can be easily defined by using the capabilities of these languages.

Our first implementation will be based on the NMOS technology. The other components of the system, like switch-level simulator, PLA generator, and so on, can be the same used in the ALI2 system.

5. References

[1] Kernighan, B. and Cherry, L.
A System for Typesetting Mathematics. Communications of the ACM, March 1975.

[2] Lengauer, T. and Mehlhorn, K.
HILL - Hierarchical Layout Language, A CAD System for VLSI Design. TR A82/10, FB 10, Universität des Saarlandes, Saarbrücken, West Germany, 1982.

[3] Mata, J.
Solving Systems of Linear Equalities and Inequalities Efficiently. Princeton University, November 1983.

[4] Lipton, R.J., North, S.C., Sedgewick, R., Valdes, J., Vijayan, G.
VLSI Layout as Programming. ACM Trans. on Programming Languages and Systems, July 1983.

[5] Lipton, R.J., Sedgewick, R., Valdes, J.
Programming Aspects of VLSI. Proc. 9th Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982.

[6] Sastry, S. and Klein, S.
PLATES: A Metric-Free VLSI Layout Language. Proc. of the 1982 Conference on Advanced Research in VLSI, MIT, January 1982.

[7] Vijayan, G.
Design, Implementation, and Theory of a VLSI Layout Language. Ph.D. Thesis, Princeton University, August 1983.

[8] Vuillemin, J. and Guibas, L.
On Fast Binary Addition in MOS Technologies. Proc. of the IEEE International Conference on Circuits and Systems, New York, September 1982.

A Fast Tally Structure and Applications to Signal Processing†

Peter R. Cappello

Department of Computer Science
University of California
Santa Barbara, California 93106

Kenneth Steiglitz

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

ABSTRACT

We describe the design, layout, and simulation of a recursively defined VLSI chip, using a constraint-based, procedural layout language. We use as an example the problem of counting the number of 1's in a set of $(B-1)$ input bits, where B is a power of 2. A regular, recursive structure, called a unary-to-binary converter (UBC(B)), tally circuit, or parallel counter, is described, based on the original design of Swartzlander. Area from the CIF plots and worst-case delay from simulations are given for 5 instantiations of the circuit, for $B = 4, 8, 16, 32,$ and 64. The results verify the expected asymptotic behavior of the implementation as a function of B.

The high-level, procedural approach leads to a succinct and parameterized description of the circuit. Verification and simulation of different versions of the circuit is much easier than with the conventional, hand-layout approach.

1. Introduction

The purpose of this paper is to describe the architectural design, layout, and simulation of a recursively defined VLSI chip. The emphasis will be on the design methodology, which relies on the VLSI layout language CLAY (and its predecessor ALI) developed at Princeton [7,8]. We hope to illustrate the following points:

1) VLSI design and layout can be viewed at the highest level as a programming task;

2) The procedural approach leads to parameterized designs, and these parameters can be bound late in the overall design process;

3) The parameterization of a circuit layout allows a clear evaluation of the functional dependence of speed, area, and power on critical circuit parameters, such as number of bits, wire lengths, and pullup and pulldown sizes.

4) The notion of regularity in chip layout means in general that the circuit has a succinct and hierarchical description, and includes more general structures than simple linear or two-dimensional arrays of elements.

The approach lends itself particularly well to the building of parameterized libraries (instead of libraries of rigid CIF cells), and ultimately can lead to a high-level language for the design of custom signal processing chips: a silicon compiler.

The particular computational element that we will describe is known variously as a *tally circuit*, *parallel counter*, or *unary-to-binary converter (UBC)*, and appears to have been first described by Swartzlander [1]. Applications to merged arithmetic and multiplier design are described in [2,3], and asymptotic analysis of the time and space requirements for the tally circuit and various of its applications is described in [4].

2. The Recursive Structure

The problem we want to consider is that of counting the number of 1's on a set of $(B-1)$ input lines, where B is an integer power of 2. We will refer to the circuit that performs this task as *unary-to-binary converter with parameter B*, or UBC(B). It has a $\log_2 B$-bit, bit-parallel output. We will consider here only an unclocked, combinational circuit, for minimum latency, but internal latching will increase the throughput at the expense of latency, area, and circuit complexity. (See, for example, [5].)

The basic recursive structure we use for solving this problem is shown in Figure 1. Inductively, we assume that the inputs are applied at the bottom, and the outputs appear at the right. To construct UBC(B), we place an instance of UBC(B/2) on the left, another instance on the right (mirror-reversed), and we add the two sets of outputs in a spine of one-bit full adders (FA's) that constitutes a $(\log_2 B - 1)$-bit full adder. The outputs of the spine adder must be routed over the right UBC(B/2). The basis of the recursion is simply UBC(4) = FA, a one-bit full adder (with 3 inputs and 2 outputs). If $N(B)$ is the number of FA's in UBC(B), then $N(B) = 2N(B) + \log B - 1$, with $N(4) = 1$; which leads to the solution $N(B) = B - \log B - 1$. Similarly, the delay $D(B)$ of UBC(B) satisfies $D(B) = D(B/2) + 2$, with

†This work was supported by NSF Grants ECS-8120037 and ECS-8307666, U. S. Army Research-Durham Grant DAAG29-82-K-0095, and DARPA Contract N00014-83-K-0249.

IEEE 1984 International Conference
on Acoustics, Speech, and Signal Processing
San Diego, California
March 19-21, 1984

$D(4) = 1$ full-adder delay, so $D(B) = 2\log_2 B - 3$ full-adder delays.

The implementation of the circuit is essentially a C program of about 300 lines, which specifies the entire chip down to the level of pullup and pulldown cells. It is translated into linear constraints and then into a CIF file by the CLAY [7,8] compiler developed at Princeton. About one-third of the description is devoted to the full-adder cell, which uses the random-logic design described in [6]. Most of the rest of the code consists of wire declarations and simple wire-routing cells. The language itself specifies only relative orientation of wires and cells (left-to-right, top-to-bottom), and not absolute spacing. The layout is generated from constraints determined by a table of design rules, so that scale changes and design-rule variations do not necessitate re-design.

3. Five Instantiations

At the time this is being written (December 1983), the design has been carried through to the level of CIF plots for $N = 4, 8, 16, 32$, and 64. Each design has been tested for design-rule violations with the Berkeley tool LYRA[9], and each design has had its worst-case delay path evaluated by the Berkeley tool CRYSTAL [9]. In addition, the smaller designs have been verified at the gate-logic level with a switch-level simulator. We plan to submit the chips for fabrication in nMOS with $\lambda = 2\mu$, after pad-routing is added.

We want to emphasize the fact that the only difference in the CLAY descriptions for the 5 cases of UBC(B) is the change of the one parameter B. The width of the power and ground lines is calculated at each level of the recursion, allowing 1μ/ma, or the minimum wire width 3λ, whichever is larger. Since the description is completely procedural, any circuit-size parameter can be made a function of the level of recursion, or any other program variable. All wire routing and placement is done automatically, without an interactive display.

The CIF plots for the first 4 instantiations are shown in Figure 2, one below the other, each to half the scale of the previous.

4. Timing-Simulation Results

The CRYSTAL[9] simulator was used throughout as a guide for inverter sizing. The fulladder has one 3-input NAND gate, six 2-input NAND gates, two inverters, and four pullups, and these were sized by trial and error, following the design methodology and conventions of Mead and Conway [10]. Future work will be directed towards the automation of this process, which becomes much easier when a high-level procedural description of the circuit is used. With $\lambda = 2\mu$ nMOS parameters, the present full adder has a worst-case path delay of 36.2 ns, takes area of 9900 λ^2, and has an estimated worst-case power dissipation of about 2 mw. The design is uniform at all levels of the recursion, although the output capacitive loading increases at higher levels because of the longer wires; clearly some speedup for the same power could be achieved by increasing the size of the output stages at higher levels, but this has not yet been studied.

Ideally, we would want the delay to be close to $D(B)*(36.2)$ ns. The extent that the delay is greater than this indicates that the long wire routing is slowing down the output stages of some of the full adders. The simulation results show this effect clearly. The measure delay/$D(B)$, the delay per full-adder stage, is listed below:

B	delay / $D(B)$, ns
4	36.2
8	35.2
16	36.1
32	39.3
64	45.4

We see that the long-wire loading becomes significant only for $B = 64$, which is an extreme case -- the last stage output wires for $B = 64$ are about 5 mm long, and the entire circuit is too wide to fabricate. UBC(32) has a worst-case delay (CRYSTAL) of 275.1 ns, has 38 pins, and represents the largest instance of UBC that we plan to fabricate and test.

The structure has the desirable property that the fanout is constant for all the full-adder cells. This enables us to keep the delay per full-adder stage almost constant for $B \leq 32$. If the fanout did grow, however, it would be easy to make the driver sizes a function of the recursion level.

5. Space Efficiency of the Layouts

The layout was constructed in a highly disciplined and conservative way: for example, conventional inverters are used, the pullups are straight and no wires cross them, metal wires are not run on top of polysilicon or diffusion signal wires. No hand-packing or hand-routing was used. Thus, we can expect to pay some penalty in terms of area utilization and speed over conventional interactive layout. In return, we gain in getting a complete parameterization of the design, and in the ability to postpone design decisions until the structure is complete. For example, the inter-adder signal wires were all changed from diffusion to polysilicon at the last step, with only a minor edit of the CLAY program. Similarly, the power and ground wires were sized at a very late stage. The situation is quite analogous to programming in a high-level block-structured language, as opposed to hand-coding in machine language. In many situations we are willing to give up a factor of of 2, say, in performance if the design time and flexibility of the result is greatly improved. We may also be able to gain back some performance because of the ease with which parameter optimization can be carried out. On the other hand, there will always be situations where hand optimization is necessary.

The width of UBC(B) is ideally proportional to B, and the height proportional to $\log B$. The actual values obtained in the layouts are shown below, normalized to a single full adder:

B	width / B	height / $\log B$
4	1	1
8	1.69	1.67
16	2.06	1.90
32	2.27	2.07
64	2.39	2.30

Thus, the inter-adder wiring contributes a factor of about 2 in both height and width in this range of B,

which means that about one-quarter of the area is filled with the full-adder logic. (For reference, the full adder fits in a 163λ × 61λ rectangle.)

The wide, low profile may be an advantage or a disadvantage, depending on the application. In the merged arithmetic structures it seems desirable to have a long, thin UBC which can collect partial products from an array. Packing schemes other than the one shown in Figure 1 are of course possible, and each has its own aspect ratio. The study of the relationship between packing schemes and aspect ratio is an interesting research question in itself.

The estimated maximum steady-state power dissipation is 2.38 *watts / cm*^{2} for UBC(8), and decreases to 1.09 *watts / cm*^{2} for UBC(64). Again, the parameterization of the circuit makes it easy to trade speed for power. If a fabricated design is too hot, the re-sizing of the inverters is a relatively minor programming change for the next generation design.

6. Conclusions

We have outlined a high-level, procedural approach to the design and layout of custom VLSI chips for digital signal processing, using the CLAY language[7,8]. The unary-to-binary converter was used as an example, and instantiations for 3, 7, 15, 31, and 63 input bits have been described. The worst-case delay from timing simulations and the area utilization of the layouts have been given.

In many ways the approach resembles programming in a high-level, block-structured language, as opposed to hand-coding machine language. Thus, we are able to study the effects of varying circuit parameters by simple program changes, after the design is structurally complete. Ultimately, we hope to take advantage of this circuit representation to study the automatic optimization of structural parameters.

7. Acknowledgements

We want to thank the VLSI group at Princeton for support of all kinds, especially A. S. LaPaugh, R. J. Lipton, J. Mata, S. C. North, D. L. Souvaine, and J. Valdes.

8. References

[1] E. E. Swartzlander, Jr., "Parallel Counters," *IEEE Trans. on Computers*, Vol. C-22, No. 11, pp. 1021-1024, Nov. 1973.

[2] --, "Merged Arithmetic," *ibid*., Vol. C-29, No. 10, pp. 946-950, Oct. 1980.

[3] --, "The Quasi-Serial Multiplier," *ibid*., Vol. C-22, No. 4, pp. 317-321, April 1973.

[4] P. R. Cappello K. Steiglitz, "A VLSI Layout for a Pipelined Dadda Multiplier," *ACM Trans. on Computer Systems*, Vol. 1, No.2, pp. 157-174, May 1983.

[5] P. R. Cappello, A. S. LaPaugh, K. Steiglitz, "Optimal Choice of Intermediate Latching to Maximize Throughput in VLSI Circuits," *Proc. 1983 IEEE International Conf on Acoustics, Speech, and Signal Processing*, April 14-16, 1983, pp. 935-938; and *IEEE Trans on Acoustics, Speech, and Signal Processing*, in press.

[6] W. C. Holton, "The Large-Scale Integration of Microelectronic Circuits," *Scientific American*, Vol. 237, No. 3, pp. 82-94, September 1977.

[7] S. C. North, "Molding Clay: A Manual for the CLAY Layout Language," VLSI Memo #3, EECS Department, Princeton University, Princeton, N. J., July 1983.

[8] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, G. Vijayan, "VLSI Layout as Programming," *ACM Tans on Programming Languages and Systems*, July 1983.

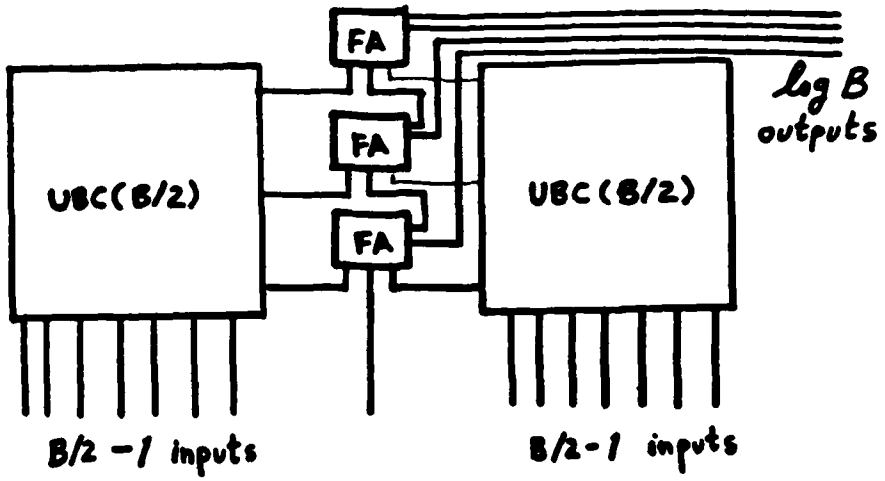
[9] R. N. Mayo, J. K. Ousterhout, W. S. Scott, "1983 VLSI Tools," Report No. UCB/CSD 83/115, Computer Science Division (EECS), University of California, Berkeley, Calif., March 1983.

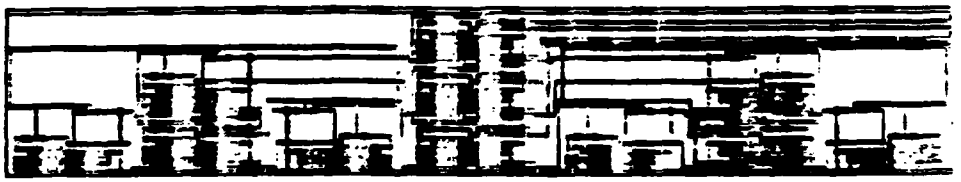
[10] C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Co. Menlo Park, Ca., 1980.

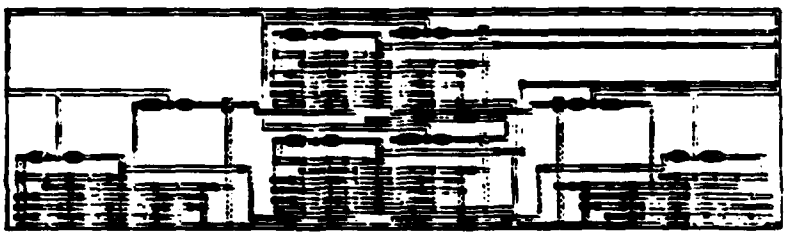
Figure Captions

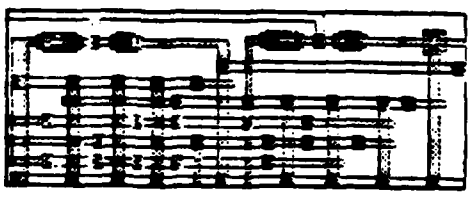
Fig. 1 Recursive definition of the structure of UBC(B).

Fig. 2 CIF plots for the cases B = 4, 8, 16, 32. Each plot is to half the scale of the preceding.


FA
FA
FA
UBC(B/2)
UBC(B/2)
log B
outputs
B/2 -1 inputs
B/2-1 inputs







SOLVING SYSTEMS OF LINEAR EQUALITIES AND INEQUALITIES EFFICIENTLY

José M. Mata

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

Technical Report #318
March 1984

SOLVING SYSTEMS OF LINEAR EQUALITIES AND INEQUALITIES EFFICIENTLY

José M. Mata

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

Abstract

In many applications, like VLSI layout systems, we have to solve a system of linear constraints on two variables. Usually these systems involve millions of variables and constraints. If we have only equations of the form $x_i - x_j \geq d$ $(d>0)$, we can use the topological sort algorithm, that is linear in time and space complexity. Allowing also constraints of the form $x_i - x_j = e$ $(e>0)$ gives more power, but affects the efficiency in solving. Algorithms for the single-source shortest path problem or for the linear programming problem can be used, but they are not efficient enough for the size of our problem.

We present here an algorithm to solve systems of equations of the form $x_i - x_j \geq d$ $(d>0)$ and $x_i - x_j = e$ $(e>0)$, with time complexity $O((n_i+n_e+v)n_e)$, where n_i = number of inequalities, n_e = number of equalities, and v = number of variables. It is specially efficient when the number of equalities is small compared to the number of inequalities.

1. Introduction

In many computer aided design systems for integrated circuit design the specification of a layout is internally represented by a set of geometric constraints on the coordinates of the layout components [5] [7] [8] [11] [12] [14]. In its simplest form these geometric constraints are linear inequalities between pairs of coordinates, and the x and y coordinates are independent. We can also have equivalence between two coordinates.

The geometric constraints relating the coordinates of the layout components are of the form:

$$\begin{array}{l} x_i - x_j \geq d \\ x_i = x_j \end{array} \qquad (d > 0, \textit{integer}) \qquad (1)$$

The number of layout components in usual circuit designs is of the order of 10^6, which correspond to the same order of variables and constraints. The efficiency of solving this system of equations is important; that is one of the reasons for choosing so simple equations to represent the layout.

The equations of the form $x_i = x_j$ can be dealt with in a preprocessing step, by making x_i and x_j the same variable. This renaming can be done by using the union-find algorithm [13]. So, from now on we will ignore such equations.

This work was supported in part by NSF Grant MCS-8004490, DARPA Contract N00014-82-K-0549, ONR Grant N00014-83-K-0275, and CAPES-Brazil.

Let's consider equations of the form $x_i - x_j \geq d$. d > 0 implies that $x_i > x_j$. The relation > is transitive, asymmetric, and irreflexive, so it establishes a partial order on the set of variables x . We can then make use of the topological sort algorithm [4] [10] to find a solution for the set of constraints if there is one, by computing the length of the critical path to each node in a directed acyclic graph. This algorithm has time and space complexity O(n+v), where n = number of inequalities, and v = number of variables.

The algorithm builds a weighted directed graph G = (V,E) from the set of inequalities I_x as follows:

V = set of variables
$E = \{ (x_j, x_i, d) \mid x_i - x_j \geq d \in I_x \}$

We use the notation (a,b,w) to denote a directed edge from a to b with weight w. Also, by length of a path we mean the sum of the weights of the edges in the path.

However, there are two problems. First, as the space complexity of the topological sort algorithm is O(n+v), for large layouts (large number of variables and inequalities) we run into memory problems. Second, sometimes we want to include in the layout some pieces of fixed size, and it is not possible to specify that using only inequalities of the form $x_i - x_j \geq d$ (d > 0).

The solution to these problems is to allow constraints of the form $x_i - x_j = e$ ($e > 0$, *integer*). This allows the specification of pieces of fixed sizes in the layout. By constructing the layout hierarchically (creating small parts of the layout and using them as pieces of fixed size in the next level of the hierarchy), will solve the problem of large layouts.

So, our problem now is to solve efficiently a set of constraints of the form:

$$\begin{aligned} x_i - x_j &\geq d \quad (d > 0, \textit{integer}) \\ x_i - x_j &= e \quad (e > 0, \textit{integer}) \end{aligned} \tag{2}$$

The number of variables and inequalities can be huge, but for our application the number of equalities is small, and it can even be limited. In the ALI2 layout system [6] [7] [13], each equality corresponds to a piece of fixed size introduced. So, in the specification of the layout it is possible to control the number of equalities generated, keeping it small (less than 100, usually). It is also possible to control the number of inequalities by constructing the layout hierarchically.

Note that the equality $x_i - x_j = e$ can be replaced by the inequalities $x_i - x_j \geq e$ and $x_i - x_j \leq e$.

Currently known methods for solving systems of linear equalities and inequalities have time complexities which are polynomial of high degree, and are not suitable for handling large number (millions) of variables and constraints.

One approach is to view this problem as a special case of linear programming, with only 2 variables per inequality. In this case, a typical constraint has the form

$$a\, x_i + b\, x_j \leq c \quad (a,b,c \textit{ are rational numbers}) \tag{3}$$

Algorithms for this problem are considered in [1] [9].

Other approach is to use single-source shortest path methods [2] [3] [6] on unrestricted graphs, or equivalently considering inequalities of the form

$$x_i - x_j \leq c \quad (c \textit{ integer}) \tag{4}$$

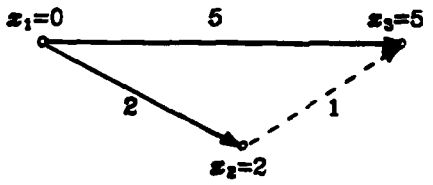
We can see that both the linear programming problem with 2 variables per inequality and the single-source shortest path problem are more general than our problem of solving a system of equalities and inequalities. In our problem the coefficient of the variables in the equations is 1, which is a particular case of the linear programming problem.

In order to use the shortest path method we construct a directed graph from our system of equations: $x_i - x_j \geq d$ corresponds to an edge $(x_j, x_i, -d)$, and $x_i - x_j = e$ corresponds to edges $(x_j, x_i, -e)$ and (x_i, x_j, e). Shortest path methods work on unrestricted graphs, while the graph corresponding to our system of equations will have edges of positive weight only in conjunction with an edge of negative weight, that is, if (a, b, c) exists, $c > 0$, then $(b, a, -c)$ exists.

So, it should be possible to find better algorithms to solve systems of equalities and inequalities of the form (2).

2. The algorithm

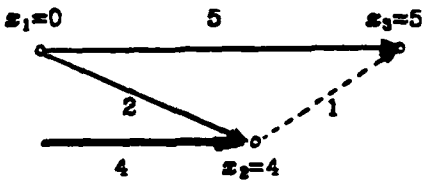
We want to use the topological sort algorithm, since it is linear in terms of time and space complexity. If we replace an equality $x_i - x_j = e$ by the inequality $x_i - x_j \geq e$, we may get an answer that is not a solution to the equality. A situation where the equality is not satisfied is for example:



$x_2 - x_1 \geq 2$
$x_3 - x_1 \geq 5$
$x_3 - x_2 = 1$

As our topological sort computes the length of the critical path to every node, the value obtained for x_i is the minimum value that x_i can get. So, x_j should be at least the value of x_i minus e.

Our approach is to introduce a new constraint $x_j \geq (value\ of\ x_i) - e$ for each unsatisfied equality $x_i - x_j = e$, and do the topological sort again.



new constraint: $x_2 \geq 4$

For theoretical purposes, we can assume that there is a source node s. The constraint $x_2 \geq 4$ would correspond to $x_2 - s \geq 4$. We also assume that there are no constraints of the form $x_i - x_i \geq d$ or $x_i - x_i = e$, since they can be detected at input time.

We claim that there is a solution to the system of equalities and inequalities if and only if each time we introduce a constraint on x_j and repeat the topological sort, at least one equality will be satisfied and will remain satisfied in subsequent steps.

Note that if an equality $x_i - x_j = e$ is not satisfied then the value of x_j must be less than the value of x_i minus e. The effect of introducing a constraint $x_j \geq (value\ of\ x_i) - e$ is to move x_j up; consequently, nodes that depend on x_j

may move up.

The algorithm can be stated as follows:

Algorithm A:

A1. Construct digraph G=(V,E):
$V = s \cup \{ x_k \mid x_k$ *appears on an equality or inequality* $\}$
$E = \{ (x_i, x_k, w) \mid x_k - x_i \geq w$ *or* $x_k - x_i = w$ *are constraints* $\}$

A2. Sort G topologically, computing the critical path to each x_k;
if cycle **then** "no solution";

A3. **while** (unsatisfied equalities) **and**
(not all unsatisfied equalities in a previous step were unsatisfied in subsequent steps, not necessarily the same) **do**

for each unsatisfied equality $x_i - x_j = e$
introduce $x_j - s \geq$ (*value of* x_i) $- e$;

sort G topologically, computing critical paths;
endwhile;

A4. **if** (unsatisfied equalities) **then** "no solution".

The second condition in the *while* statement is equivalent to saying that there was some progress, or that at least one of the unsatisfied equalities in previous steps was fixed up.

By saying that all unsatisfied equalities in a step are unsatisfied in subsequent steps, we mean that each of these unsatisfied equalities will be unsatisfied again later on, not necessarily all at the same time. Let's express this in a different way. Let's call U_s the set of unsatisfied equalities at step s, and $R_{s,t}$ the set of all unsatisfied equalities from steps s to t, including s and t ($t \geq s$).

$$R_{s,t} = U_s \cup U_{s+1} \cup \cdots \cup U_t$$

If $U_s \subseteq R_{s+1,t}$, $t > s$, then at step t we can conclude that all unsatisfied equalities at step s were unsatisfied again.

3. Correctness

We want to show that if there is a solution then in the execution of algorithm A eventually all equalities (and inequalities) will be satisfied, and if there is no solution then all unsatisfied equalities in a step will be unsatisfied later on. This guarantees that the algorithm always terminates with the correct answer. These ideas are expressed in the following theorem:

Theorem 1:

There is a solution to the system of equations if and only if not all unsatisfied equalities in a step are unsatisfied in subsequent steps.

The proof of this theorem is contained in the next three lemmas.

Lemma 1:

If not all unsatisfied equalities in a step are unsatisfied in subsequent steps then there is a solution to the system of equations.

Proof:

If not all unsatisfied equalities in a step are unsatisfied in subsequent steps, that means that at least one of the unsatisfied equalities in that step will be satisfied in all subsequent steps. Let's assume that there are n_e equalities. In the first step there are at most n_e unsatisfied equalities, in the second step at most $n_e - 1$, and so on. So, after at most n_e steps all equalities will be satisfied, which means that there is a solution.

We can also derive from this proof that in at most n_e steps we can find a solution, or detect that the equalities remained unsatisfied.

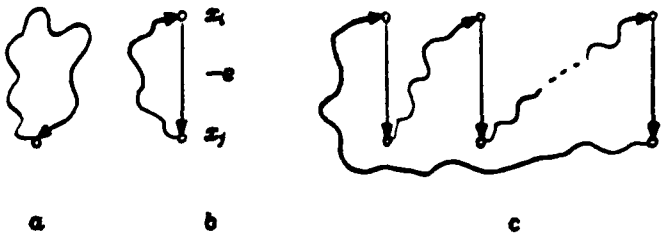
Now we have to show that if the equalities remained unsatisfied then there is no solution. We will need an intermediate result to do that. Let's consider the original digraph G. Let's construct G' by adding the edge from x_i to x_j labeled $-e$ for each equality $x_i - x_j = e$. This corresponds to $x_i - x_j \leq e$, or $x_j - x_i \geq -e$.

Lemma 2:

If there is a cycle in G' of length greater than zero then there is no solution to the system of equations.

Proof:

The cycle can have 0, 1 or more edges of negative weight.



a) If the cycle doesn't involve cycles of negative weight, that means a cycle in G, that is, no possible solution.

b) If the cycle involves one edge of negative weight then the endpoints of this edge are variables involved in an equality. If the length of the cycle is greater than zero, then there is a path from x_j to x_i of length greater than e, that is, the length of the critical path from x_j to x_i is greater than e, so $x_i - x_j = e$ is not possible.

c) An edge from x_i to x_j labeled $-e$ means that wherever x_i is, x_j should be below x_i separated by the exact distance e. Let's take a negative edge (x_i, x_j) in the cycle containing more than one negative edge. Again, the length of the critical path from x_j to x_i, considering negative edges, is greater than e, so $x_i - x_j = e$ is not possible.

Lemma 3:

If all unsatisfied equalities in a step are unsatisfied in subsequent steps, then there is a cycle in G' of length greater than zero.

Proof:

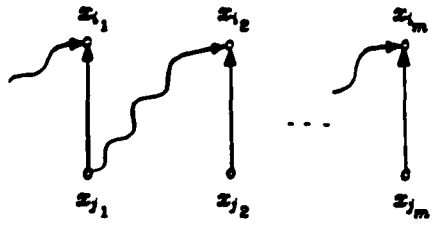
Let's first consider the situation in which no x_k appears in two equalities, that is, the equalities are disjoint.

When an equality $x_i - x_j = e$ is unsatisfied we introduce a constraint $x_j \geq (value\ of\ x_i) - e$; if after the next solving step x_i has the same value as before then the equality will be satisfied (remember that there is an edge from x_j to x_i labeled e). If the equality is unsatisfied in a subsequent step (not necessarily the next), that means that x_i moved up.

Let's assume that at step s there are m unsatisfied equalities, and at step t ($t > s$) we conclude that all those equalities were unsatisfied again. That means that each x_i involved in one of the m equalities $x_i - x_j = e$ moved up; in other words, the critical path to x_i goes through some new edge corresponding to a constraint introduced.

If x_i goes up because of a constraint on x_j ($x_i - x_j = e$), then the critical path from x_j to x_i is greater than e, which corresponds to a cycle of length greater than zero in G'.

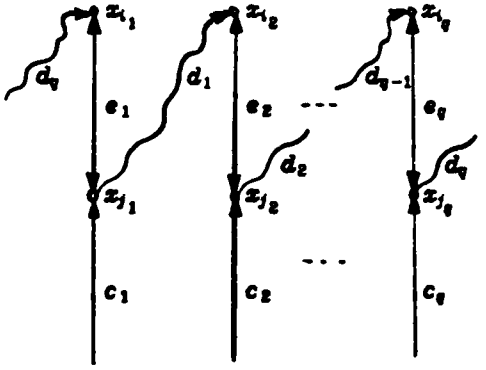
The only possibility is that x_i goes up because of a constraint introduced for other equality.



Note that the effect might not be immediate, that is, a constraint on x_{j_r} forces constraints on other x_j's, which finally forces x_{i_s} to move up.

So, in G' there is a (critical) path from x_{j_r} to x_{i_s}, $r \neq s$, for $1 \leq s \leq m$. We first want to show that this corresponds to a cycle in G'. Let's consider the directed graph constructed by taking as nodes each pair (x_{i_s}, x_{j_s}), $1 \leq s \leq m$, and drawing an edge from (x_{i_r}, x_{j_r}) to (x_{i_s}, x_{j_s}) if there is a (critical) path from x_{j_r} to x_{i_s} in G', $r \neq s$. The in-degree of each node in this new graph is at least 1. If there was no cycle in this graph there should be a node with in-degree 0 where we could start a simple path. As such node doesn't exist, we conclude that there is a cycle in this graph, and correspondingly a cycle in G'.

Let's consider one cycle, with q vertices.



d_k = length of the path from x_{j_k} to $x_{i_{k+1}}$.

c_k = value of c in the last constraint $x_{j_k} \geq c$ introduced for x_{j_k}.

Because of our assumption that x_{i_q} goes up because of a constraint on x_{j_r}, $r \neq s$, the critical path to x_{i_q} goes through x_{j_r}, and we can write for the last step (step t):

$$c_1 + d_1 \geq c_2 + e_2$$

$$c_2 + d_2 \geq c_3 + e_3$$

$$\vdots$$

$$c_{q-1} + d_{q-1} \geq c_q + e_q$$

$$c_q + d_q \geq c_1 + e_1$$

There is at least one unsatisfied equality in the last step, so at least one of the above $\geq$ should be $>$. Let's consider

$$c_p + d_p > c_{p+1} + e_{p+1}$$

Summing up:

$$(\sum c - c_p) + (\sum d - d_p) \geq (\sum c - c_{p+1}) + (\sum e - e_{p+1})$$

$$\sum d - (c_p + d_p) \geq \sum e - (c_{p+1} + e_{p+1})$$

$$c_p + d_p > c_{p+1} + e_{p+1} \quad ===> \quad \sum d > \sum e$$

Considering the digraph G', this implies that there is a cycle of length greater than zero.

In the situation where the equalities are not disjoint, the proof of lemma 3 proceeds the same way. If x_k appears in two equalities we replace x_k by x_{k_1} and x_{k_2}, and assume there is a path from x_{k_1} to x_{k_2}, and from x_{k_2} to x_{k_1}, with weight 0.

This way we can consider the equalities disjoint, and the proof of lemma 3 holds.

Lemmas 2 and 3 imply that if there is a solution to the system of equations then not all unsatisfied equalities in a step are unsatisfied in subsequent steps. So, the proof of theorem 1 is completed.

4. Complexity

Let n_i = number of inequalities, n_e = number of equalities, v = number of variables. The topological sort has time and space complexity = $O(n_i + n_e + v)$. Checking if equalities remain unsatisfied takes time $O(n_e)$ and space $O(n_e^2)$. The *while* loop in algorithm A is repeated at most n_e times. So, the worst case complexity for algorithm A is:

Time complexity $= O((n_i + n_e + v)\, n_e)$

Space complexity $= O(n_i + n_e^2 + v)$

If the number of equalities is very small compared to the number of inequalities and variables then the algorithm is almost linear in the number of inequalities and variables.

5. Characterization of the solution

The system of equalities and inequalities may admit an infinite number of solutions. We claim that if there is a solution then algorithm A finds the solution with mininum possible value for each x_k.

Theorem 2:

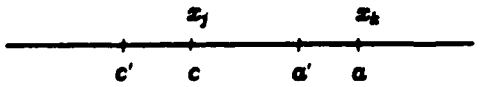
The solution found by algorithm A has the minimum possible value for each variable that satisfies all equalities and inequalities.

Proof:

In order to construct the graph upon which algorithm A works, the equality $x_i - x_j = e$ is replaced by the inequality $x_i - x_j \geq e$. The value obtained for each variable, after the topological sort, is the minimum value that satisfies the inequalities, since our topological sort computes the length of the critical path to every node.

When we introduce a constraint $x_j \geq c$, $c = (\textit{value of } x_i) - e$, for $x_i - x_j = e$, that means that x_j must have the value at least c, that is, there is no solution with $x_j < c$.

Let's assume that the solution found by algorithm A is not the minimum solution. Then there should be some variable x_k with value a in the solution, $x_k = a$, such that there is other solution with $x_k = a'$, $a' < a$.



x_k moved up because of a constraint introduced, that is, in our graph the critical path to x_k goes through some edge (s, x_j, c), corresponding to $x_j \geq c$. As we didn't change the length of the path from x_j to x_k, the solution with $x_k = a'$ should have $x_j = c'$, $c' < c$, which is not possible. By contradiction, we conclude that the solution found by algorithm A is the minimum solution.

6. Implementation considerations

The topological sort doesn't have to be done many times. If we split it into sorting and assigning values, we can do the sort only once, and the assignment of values each time we introduce new constraints. This assignment of values involves traversing the edge list for nodes that have their values updated. We can start this process with the first node in the ordered list that was involved in a new constraint.

In order to check if unsatisfied equalities remain unsatisfied we can have a bit vector for each step describing which equalities are not satisfied at that step. Equalities that are unsatisfied in subsequent steps have their corresponding bits reset. If the bit vector becomes zero then all unsatisfied equalities in that step remained unsatisfied.

The algorithm has been implemented in the ALI2 layout system [14], and is faster than we expected.

7. Conclusions

We presented an algorithm to solve systems of equations of the form $x_i - x_j \geq d \quad (d > 0)$, and $x_i - x_j = e \quad (e > 0)$, with time complexity $O((n_i + n_e + v)n_e)$. This algorithm is specially useful when the number of equalities n_e is small compared to the number of inequalities n_i, as it is the case for many constraint-based VLSI layout systems.

With this algorithm to solve the set of constraints, we can construct a truly hierarchical layout: we can have hierarchy not only at the specification level but also at the constraint level. This allows the construction of large layouts without having to deal with large sets of constraints. It also allows the introduction of pieces of fixed size, produced by other layout tools.

Acknowledgements

Thanks to Prof. Andrea LaPaugh and to Alfred Huang for helpful discussions, comments, and suggestions.

References

[1] Aspvall, B. and Shiloach, Y.
A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two Variables per Inequality. SIAM Journal on Computing, Vol. 9, No. 4, Nov 1980.

[2] Bloniarz, P.
A Shortest-Path Algorithm with Expected Time $O(n^2 \log n \log^* n)$. SIAM Journal on Computing, Vol. 12, No. 3, Aug 1983.

[3] Johnson, D.
Efficient Algorithms for Shortest Paths in Sparse Networks. Journal of the ACM, vol. 24, n. 1, Jan 1977.

[4] Knuth, D.
The Art of Computer Programming, vol. 1. Addison-Wesley, 1971.

[5] Lengauer, T. and Mehlhorn, K.
HILL - Hierarchical Layout Language, A CAD System for VLSI Design. TR A82/10, FB 10, Universität des Saarlandes, Saarbrücken, West Germany, 1982.

[6] Lengauer, T.
On the Solution of Inequality Systems Relevant to IC-Layout. Proc 8th Conf. on Graphtheoretic Concepts in Computer Science. Munich, West Germany, 1982.

[7] Lipton, R.J., North, S.C., Sedgewick, R., Valdes, J., Vijayan, G.
VLSI Layout as Programming. ACM Trans. of Programming Languages and Systems, July 1983.

[8] Lipton, R.J., Sedgewick, R., Valdes, J.
Programming Aspects of VLSI. Proc. 9th Symposium on Principles of Programming Languages, Albuquerque, New Mexico, Jan 1982.

[9] Megiddo, N.
Towards a Genuinely Polynomial Algorithm for Linear Programming. SIAM Journal on Computing, Vol. 12, No. 2, May 1983.

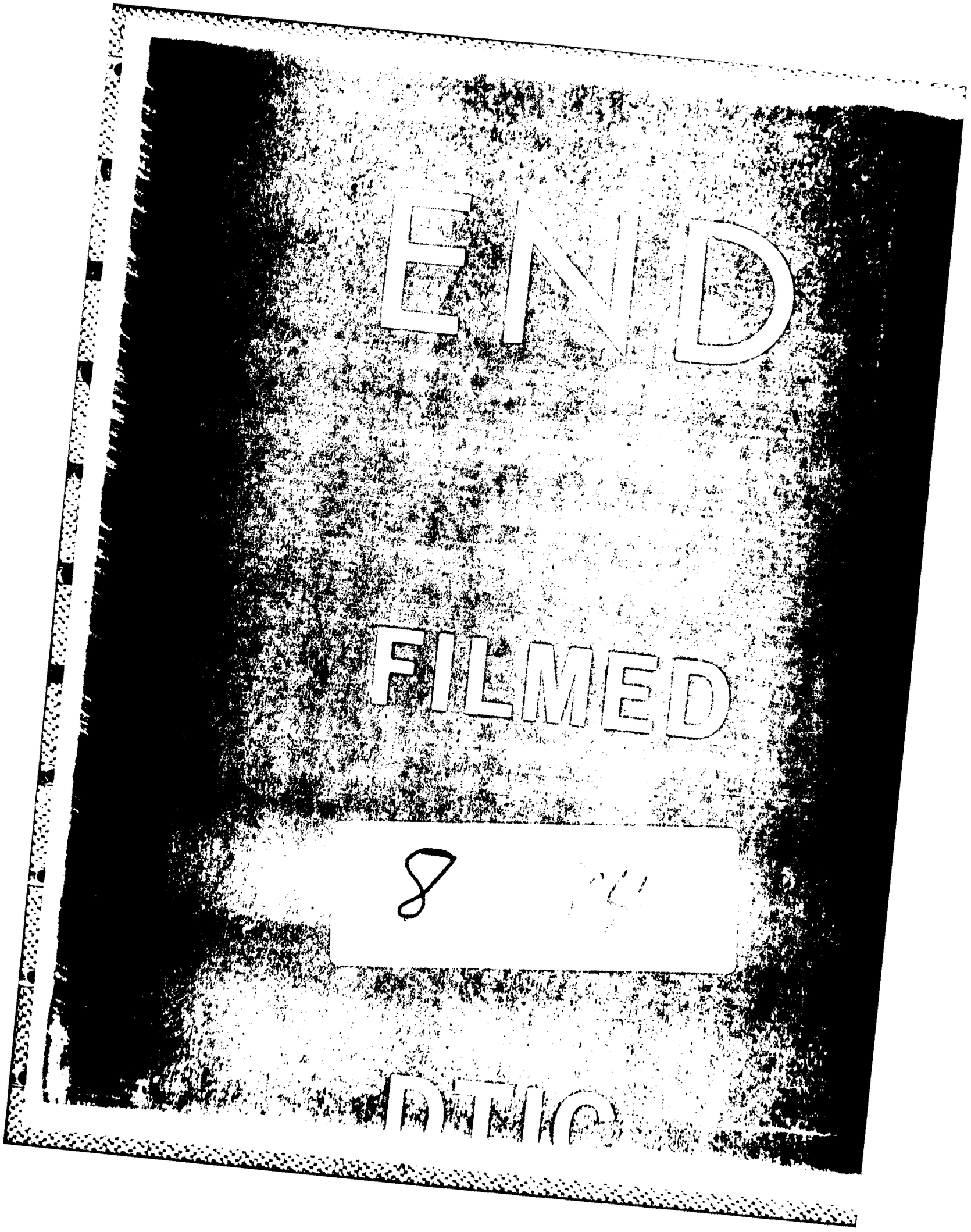
[10] Moder, J. and Phillips, C.
Project Management with CPM and PERT. Van Nostrand Reinhold Co., 1964.

[11] North, S.
Molding Clay: A Manual for the CLAY Layout Language. VLSI memo #3, Princeton University, July 1983.

[12] Sastry, S. and Klein, S.
PLATES: A Metric-Free VLSI Layout Language. Proc. of the 1982 Conference on Advanced Research in VLSI, MIT, Jan 1982.

[13] Tarjan, R.
On the Efficiency of a Good but not Linear Set Union Algorithm. Journal of the ACM, Vol. 22, No. 2, 1975.

[14] Vijayan, G.
Design, Implementation, and Theory of a VLSI Layout Language. Ph.D. Thesis, Princeton University, August 1983.


END
FILMED
8